COVER Book Cover

Systems Application Architecture

Common Programming Interface
FORTRAN Reference

Document Number SC26-4357-02

File Number S370-40

TITLE Title Page

Systems Application Architecture
Common Programming Interface
FORTRAN Reference

Document Number SC26-4357-02

EDITION Edition Notice
Third Edition (September 1990)

This edition replaces and makes obsolete the previous edition, SC26-4357-1.

This edition applies to IBM's Systems Application Architecture FORTRAN and to the following:

VS FORTRAN Version 2 Release 4, Program Number 5668-806
¦ IBM FORTRAN/400, Program Number 5730-FT1
IBM FORTRAN/2, Program Number 6280185

and to all subsequent releases and modifications until otherwise indicated in new editions. Consult the latest edition of the applicable IBM system bibliography for current product information.

Specific changes are indicated by a vertical bar to the left of the change. Editorial changes that have no technical significance are not noted. For a detailed list of changes, see "Summary of Changes" in topic CHANGES.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

A form for reader's comments appears at the back of this publication. If the form has been removed, address your comments to: IBM Corporation, Programming Publishing, P.O. Box 49023, San Jose, California, U.S.A. 95161-9023.

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

PREFACE Special Notices

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates.

Any reference to an IBM licensed program or other IBM product in this publication is not intended to state or imply that only IBM's program or other product may be used. Any functionally equivalent program which does not infringe any of IBM's intellectual property rights may be used instead of the IBM product. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering the subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

The following terms, denoted by an asterisk (*) on their first occurrences in this publication, are trademarks of the IBM Corporation in the United States and/or other countries:

AD/Cycle	OS/2
FORTRAN/2	OS/400
FORTRAN/400	RPG
IBM	SAA
MVS	Systems Application Architecture
Operating System/2	VM
Operating System/400	

CONTENTS Table of Contents

1.0 Chapter 1. Introduction

This introductory section:

- Identifies the book's purpose and audienc
- Gives a brief overview of the Systems Application Architecture* (SAA* solution
- Explains how to use the book

Subtopics

1.1 Who Should Read This Book

This book defines the SAA FORTRAN interface. It is intended for programmers who want to write applications that adhere to this definition.

This book is a reference rather than a tutorial. It assumes you are already familiar with FORTRAN programming concepts.

1.2 What the SAA Solution Is

The SAA solution is based on a set of software interfaces, conventions and protocols that provide a framework for designing and developing applications.

The SAA solution:

- Defines a common programming interface that you can use to develo applications that can be integrated with each other, and transported to run in multiple SAA environments
- Defines common communications support that you can use to connec applications, systems, networks, and devices
- Defines a common user access that you can use to achieve consistenc in panel layout and user interaction techniques
- Offers some applications and application development tools written b IBM*.

Subtopics
1.2.1 Supported Environments
1.2.2 Common Programming Interface

1.2.1 Supported Environments

Several combinations of IBM hardware and software have been selected as SAA environments. These are environments in which IBM will manage the availability of support for applicable SAA elements, and the conformance of those elements to SAA specifications. The SAA environments are the following:

MVS

- TSO/E
- CICS
- IMS

VM*/CM

Operating System/400* (OS/400*

Operating System/2* (OS/2*

1.2.2 Common Programming Interface

As its name implies, the common programming interface (CPI) provides languages, commands, and calls that programmers can use to develop applications which take advantage of SAA consistency. These applications can be easily integrated and transported across the supported environments.

The components of the interface currently fall into two general categories:

- Language
  - Application Generator
  - C
  - COBOL
  - FORTRAN
  - PL/I
  - Procedures Language
  - RPG*
- Service
  - Communications Interface
  - Database Interface
  - Dialog Interface
  - Presentation Interface
  - Query Interface
  - Repository Interface.

The CPI is defined by this and the other CPI reference books. The CPI is not in itself a product or a piece of code. But--as a definition--it does establish and control how IBM products are being implemented, and it establishes a common base across the applicable SAA environments.

Thus, when you want to create an application that can be used in more than one environment, you can stay within the boundaries of the CPI and obtain easier portability. (Naturally, the design of such applications should be done with portability in mind as well.)

A list of SAA books to help you can be found under "Related Documentation" in topic 1.5 and on the back cover of this book.

1.3 How to Use This Book

Subtopics

1.3.1 Relationship to Products

The SAA FORTRAN interface defines the elements that are consistent across the applicable SAA environments. Preparing and running programs requires the use of a FORTRAN product that implements the interface on one of these systems.

For the FORTRAN interface, these products are:

- VS FORTRAN Version 2 Release 4 (5668-806) on MVS and V
- IBM FORTRAN/400* (5730-FT1) on Operating System/40
- IBM FORTRAN/2* (6280185) on Operating System/

These products have their own books, and you will need to use those books in addition to this one. This book defines the interface elements that are common across the environments. The product books describe any additional elements, and--more importantly--explain how to prepare and run a program in that particular environment.

See "Related Documentation" in topic 1.5 for a list of those books.

1.3.2 How Product Implementations Are Designated

Because the SAA solution is still evolving, complete and consistent products may not be available yet on all the applicable systems. Some interface elements may not be implemented everywhere. Others may be implemented, but differ slightly in their syntax or semantics (how they are coded or how they behave at run time).

These conditions are identified in this book in two ways:

A system checklist precedes each interface element. If the interfac element is implemented or announced on a particular system, that column is marked with an X. If it is not, that column is blank.

For the SAA FORTRAN interface, all of the interface elements are implemented or announced for the four applicable systems:

MVS	**VM**	**OS/400**	**OS/2**
X	**X**	**X**	**X**

The FORTRAN interface definition is printed in black ink. If th implementation of an interface element in an operating environment differs from the SAA definition in its syntax or semantics, text is printed in green--as is this sentence.

1.3.3 How to Read the Syntax Diagrams

Throughout this book, syntax is described using the structure defined below.

Read the syntax diagrams from left to right, from top to bottom following the path of the line.

The --- symbol indicates the beginning of a statement.

The --- symbol indicates that the statement syntax is continued on the next line.

The --- symbol indicates that a statement is continued from the previous line.

The --- symbol indicates the end of a statement.

Diagrams of syntactical units other than complete statements start with the --- symbol and end with the --- symbol.

Required items appear on the horizontal line (the main path)

```
---STATEMENT-------required_item-------------------------------------
```

Optional items appear below the main path

```
---STATEMENT-------------------------------------------------------------
                 +-optional_item-+
```

If you can choose from two or more items, they appear vertically, in stack.

If you *must* choose one of the items, one item of the stack appears on the main path.

```
---STATEMENT------required_choice1-----------------------------------
               +--required_choice2--+
```

If choosing one of the items is optional, the entire stack appears below the main path.

```
---STATEMENT-------------------------------------------------------------
               +--optional_choice1--¦
               +--optional_choice2--+
```

An arrow returning to the left above the main line indicates an ite that can be repeated.

```
               +------------------+
                                  ¦
---STATEMENT-----repeatable_item-------------------------------------
```

A repeat arrow above a stack indicates that you can repeat the items

in the stack. Where a comma is included in the repeat symbol, it must be used between repeated items.

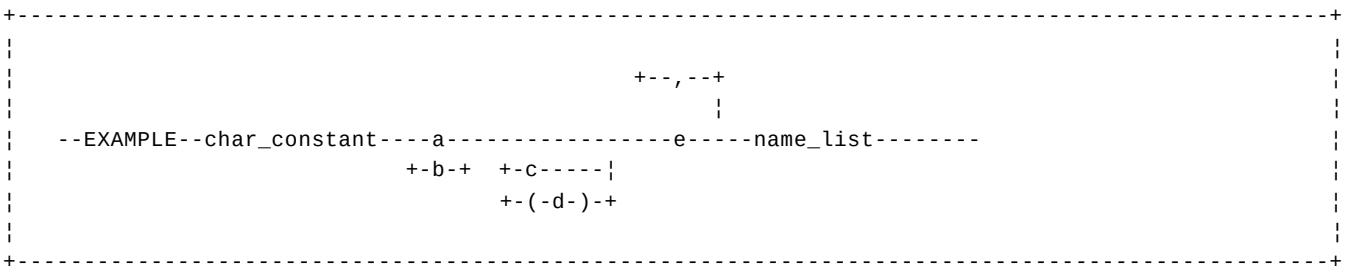
Keywords appear in uppercase (for example, **EQUIVALENCE**). They must be spelled exactly as shown.

Lowercase letters (for example *array_element_name*) represent user-supplied names or values. If one of these terms ends in *_list*, it specifies a list of the terms, where a list is a nonempty sequence of the terms separated by commas. For example, the term *name_list* specifies a list of the term *name*.

If parentheses are shown, you must enter them as part of the syntax

Items within brackets ([]) are optional. An ellipsis (...) followin an item indicates that the item may be repeated.

The following example of a fictitious statement illustrates how the syntax is used:



In the fictitious statement **EXAMPLE** you would:

Use the keyword **EXAMPLE**.

Substitute a value for **char_constant**.

Substitute a value for **a** or **b**, but not both.

Substitute a value for **c**, a value for **d**, or no value. If you substitute a value for **d**, you must include the parentheses.

Substitute at least one value for **e**. If you substitute more than one value, you must put a comma between each.

Substitute the value of at least one **name** for **name_list**.

1.4 A Note about Examples

Examples in this book help explain elements of the SAA FORTRAN language. For this purpose they are coded in a simple style. They do not attempt to conserve storage, check for errors, achieve fast execution, or demonstrate all possible uses of a language element.

1.5 Related Documentation

Subtopics
1.5.1 For the SAA Solution
1.5.2 For Implementing Products

1.5.1 For the SAA Solution

An introduction to the SAA solution in general can be found in *SAA: An Overview*, GC26-4341.

An introduction to the common programming interface can be found in *Common Programming Interface: Summary*, GC26-4675.

More detailed information on the components of the common programming interface is available in the following SAA manuals (including this one):

Application Generator Reference, SC26-4355
C Reference--Level 2, SC09-1308
COBOL Reference, SC26-4354
Communications Reference, SC26-4399
Database Reference, SC26-4348
Dialog Reference, SC26-4356
FORTRAN Reference, SC26-4357
PL/I Reference, SC26-4381
Presentation Reference, SC26-4359
Procedures Language Reference, SC26-4358
¦ *Procedures Language Level 2 Reference*, SC24-5549
Query Reference, SC26-4349
Repository Reference, SC26-4684
RPG Reference, SC09-1286.

General programming advice may be found in *Writing Applications: A Design Guide*, SC26-4362. An introduction to the use of the AD/Cycle* application development tools can be found in *AD/Cycle Concepts*, GC26-4531.

A definition of the common user access can be found in *Common User Access: Advanced Interface Design Guide*, SC26-4582, and *Common User Access: Basic Interface Design Guide*, SC26-4583.

More information on the common communications support can be found in *Common Communications Support: Summary*, GC31-6810.

Ordering Information: Contact your local IBM branch office for information on how to order the above publications. They also can be obtained through an authorized IBM dealer. The entire set of SAA publications can be ordered by specifying the bill-of-forms number SBOF-1240.

1.5.2 For Implementing Products

Subtopics

1.5.2.1 VS FORTRAN Version 2 Publications

VS FORTRAN Version 2 Language and Library Reference, SC26-4221
VS FORTRAN Version 2 Programming Guide, SC26-4222

¦*1.5.2.2 FORTRAN/400 Publications*

¦ *IBM FORTRAN/400 Language Reference*, SC21-9844
¦ *IBM FORTRAN/400 User's Guide*, SC21-9845

1.5.2.3 FORTRAN/2 Publications

IBM FORTRAN/2 Fundamentals
IBM FORTRAN/2 Compile, Link, and Run
IBM FORTRAN/2 Language Reference

1.6 Industry Standards

Systems Application Architecture FORTRAN is designed according to the specifications of the following industry standards as understood and interpreted by IBM as of September, 1987:

American National Standard Programming Language FORTRAN, ANS X3.9-1978 (also known as 1977 ANS FORTRAN)

International Organization for Standardization ISO 1539-198 Programming Languages-FORTRAN. This standard specifies the same level of FORTRAN as 1977 ANS FORTRAN. In this book, references to 1977 ANS FORTRAN are references to both standards.

American National Standard Coded Character Set--7-bit America Standard Code for Information Interchange, ANSI X3.4-1986 (also known as ASCII).

American National Standard Binary Floating-Point Arithmetic, ANSI/IEE 754-1985, with the following differences:

- Rounds to nearest mode only.
- No rounding precision mode.
- No trapping (signaling) NaNs (not a number).
- No user traps.
- Exception status flags are not supported.

The bit-manipulation intrinsic functions are based on those described in Industrial Computer System FORTRAN Procedures for Executive Functions, Process Input/Output, and Bit Manipulation, ANSI/ISA-S61.1.

1.7 Interface Definition Table

The table below lists the language elements currently in the FORTRAN interface for Systems Application Architecture.

The table indicates that all systems have an IBM licensed program announced or available that implements the language elements.

On MVS and VM, the implementing product is VS FORTRAN Version 2 Release 4 (5668-806). On Operating System/400, the implementing product is IBM FORTRAN/400 (5730-FT1). On Operating System/2, the implementing product is IBM FORTRAN/2 (6280185).

Table 1. Major Elements of the FORTRAN Interface

Interface Element	**MVS**	**VM**	**OS/400**	**OS/2**
All elements of 1977 ANS FORTRAN	X	X	X	X
IBM extensions:				
Case-insensitive source	X	X	X	X
31-character names	X	X	X	X
Underscore character (_) in names	X	X	X	X
INTEGER*2 data type	X	X	X	X
COMPLEX*16 data type	X	X	X	X
LOGICAL*1 data type	X	X	X	X
Optional length specification for INTEGER, REAL, COMPLEX, and LOGICAL	X	X	X	X
EQUIVALENCE allows association of character and noncharacter items	X	X	X	X
Data initialization in type statements	X	X	X	X
COMMON allows character and non-character items in same block	X	X	X	X
IMPLICIT NONE form of the IMPLICIT statement	X	X	X	X
Z edit descriptor	X	X	X	X
INCLUDE compiler directive	X	X	X	X
CONJG, HFIX, and IMAG intrinsic functions	X	X	X	X
Bit-manipulation intrinsic functions	X	X	X	X

+----------+

2.0 Chapter 2. Characters, Names, Lines, Statements, and Execution Sequence

This chapter describes:

- Character
- Name
- Line
- Statement
- Statement label
- Order of statements and comment line
- Normal execution sequence and transfer of control

Subtopics

2.1 Characters

The FORTRAN character set consists of **letters**, **digits**, and **special characters**:

Letters				**Digits**	**Special Characters**	
Uppercase Letters		Lowercase Letters			Characters	Names of Characters
A	N	a	n	0		Blank
B	O	b	o	1	=	Equals (equal sign)
C	P	c	p	2	+	Plus (plus sign)
D	Q	d	q	3	-	Minus (minus sign)
E	R	e	r	4	*	Asterisk
F	S	f	s	5	/	Slash
G	T	g	t	6	(	Left parenthesis
H	U	h	u	7	)	Right parenthesis
I	V	i	v	8	,	Comma
J	W	j	w	9	.	Decimal point (period)
K	X	k	x		$	Currency symbol
L	Y	l	y		'	Apostrophe
M	Z	m	z		:	Colon
					_	Underscore

An **alphanumeric character** is a letter or a digit.

In statements, lowercase letters are equivalent to their uppercase counterparts, except within:

- Character constant
- H and apostrophe edit descriptors

In statements, blanks are significant only in:

- Character constant
- H and apostrophe edit descriptor
- The count of characters permitted in a statement

You may use blanks anywhere else within a program unit to make it more readable.

The characters have an order known as a **collating sequence**, which is system-dependent. The collating sequence depends on the system's coded character set: EBCDIC on MVS, VM, and OS/400, or ASCII on OS/2.

On OS/2, the carriage return, the line feed, and the end-of-file characters must not be used as part of a character constant, an H edit descriptor, an apostrophe edit descriptor, or a comment.

2.2 Names

A **name** is a sequence of letters, digits, or underscores, the first of which must be a letter. A name identifies:

A main program, external function, subroutine, block data subprogram or common block. The maximum length of these names is 7 characters.

A variable, array, constant, argument, or statement function. Th maximum length of these names is 31 characters.

(In 1977 ANS FORTRAN, the maximum length of any name is 6 characters.)

Examples of Names:

```
XPos
shell
MAX0
TWENTY_FIVE
LongerThanSix
```

Subtopics
2.2.1 Scope of a Name

2.2.1 Scope of a Name

Each name in a program unit has a **scope**. That scope is either **global** to an executable program or **local** to a program unit, with the following exceptions:

The name of a common block in a program unit may also be the name o an array, a statement function, a dummy procedure, or a variable (but not a variable name that is also an external function name in a function subprogram).

In a function subprogram, at least one function name (on the FUNCTIO or ENTRY statement) must also be the name of a variable in that function subprogram.

Names with global scope are the name of the main program, the names of all subprograms, and the names of common blocks. All of these names have the scope of an executable program.

Names with local scope are:

Names of variables, arrays, constants, statement functions, dumm procedures, and intrinsic functions. These names have a scope of a program unit. (A name that is a dummy argument is classified as a variable, array, or dummy procedure).

Names of variables that appear as dummy arguments in a statemen function statement. These names have a scope of that statement.

Names of variables that appear as the DO-variable of an implied-D list in a DATA statement. These names have a scope of the implied-DO list.

2.3 Lines

A **line** is a sequence of 72 characters. The character positions in a line are called **columns** and are numbered consecutively 1 through 72.

There are three kinds of lines:

A **comment line** does not affect the executable program and may be used to provide documentation. It may have either of the following forms:

- C or * in column 1 and, optionally, any characters permitted in a character constant (see page 3.11) in columns 2 through 72.
- Blanks in columns 1 through 72.

An **initial line** is the first line of a statement. It is any line that is not a comment line and that contains blanks or a statement label in columns 1 through 5 and a blank or zero in column 6.

A **continuation line** is a line that continues a statement beyond its initial line. It contains blanks in columns 1 though 5, and any character from the FORTRAN character set other than blank or zero in column 6. A statement may have as many as 19 continuation lines. The END statement is the only statement that must not be continued.

2.4 Statements

Statements are used to form program units. Each statement is written in columns 7 through 72 of an initial line and as many as 19 continuation lines. Thus, a statement has a maximum of 1320 characters (20 lines x 66 characters).

Each statement is classified as executable or nonexecutable.

Table 2. Systems Application Architecture FORTRAN Statements

Statement	**Statement Group**	**Executable or Nonexecutable**
Arithmetic assignment Logical assignment Statement label assignment (ASSIGN) Character assignment	Assignment	Executable - specify actions
DIMENSION EQUIVALENCE COMMON Type: INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, CHARACTER IMPLICIT PARAMETER EXTERNAL INTRINSIC SAVE	Specification	Nonexecutable - specify the characteristics and arrangement of data
DATA	DATA	Nonexecutable - specifies the initial values of data
Unconditional GO TO Computed GO TO Assigned GO TO Arithmetic IF Logical IF Block IF, ELSE IF, ELSE, END IF DO CONTINUE STOP PAUSE END CALL RETURN	Control	Executable - specify actions
PROGRAM FUNCTION Statement function SUBROUTINE ENTRY BLOCK DATA	Program unit and procedure	Nonexecutable - classify program units, specify statement functions, and specify entry points within subprograms
READ WRITE PRINT	Input/output	Executable - specify actions

OPEN		
CLOSE		
INQUIRE		
BACKSPACE		
ENDFILE		
REWIND		
FORMAT	FORMAT	Nonexecutable - contains editing information

2.5 Statement Labels

A **statement label** is one to five digits, one of which must be nonzero. A statement is labeled by placing a statement label anywhere in columns 1 through 5 of its initial line.

The same label must not be given to more than one statement in a program unit. Blanks and leading zeros are not significant in distinguishing between statement labels. Any statement may be labeled, but only executable statements and FORMAT statements may be referred to by the use of statement labels. The statement making the reference and the statement being referenced must be in the same program unit.

2.6 Order of Statements and Comment Lines

The required order of statements and comment lines in a program unit is shown in the diagram below. In the diagram:

Statements and comment lines above a horizontal line must preced those below the line. For example, PARAMETER statements must precede DATA, statement function, executable, and END statements, and must follow PROGRAM, FUNCTION, SUBROUTINE, BLOCK DATA, and IMPLICIT NONE statements.

Vertical lines separate statements and comment lines that may b interspersed. For example, PARAMETER statements may be interspersed with comment lines and with FORMAT, ENTRY, IMPLICIT, and other specification statements.

Table 3. Order of Statements and Comment Lines

	PROGRAM, FUNCTION, SUBROUTINE, or BLOCK DATA statement			
		IMPLICIT NONE statement(1)		
			IMPLICIT statements	
		PARAMETER statements(4)	Other specification statements(4)	
Comment lines(2)	FORMAT and ENTRY(3) statements		Statement function statements(5)	
		DATA statements	Executable statements	
	END statement			

Notes:

1. The IMPLICIT NONE statement, if used, must be the only IMPLICIT statement in a program unit.

2. Comment lines may appear anywhere in a program unit before the END statement, even:

   Before a program unit's first statement
   Between an initial line and its first continuation line
   Between two continuation lines.

3. An ENTRY statement must not appear between a block IF statement and its corresponding END IF statement, or between a DO statement and the terminal statement of its DO loop.

4. Any specification statement that specifies the type of a constant's name must precede the PARAMETER statement that defines the name. A

PARAMETER statement that defines a constant's name must precede any use of the name.

5. A statement function may reference another statement function that precedes it, but not one that follows it.

2.7 Normal Execution Sequence and Transfer of Control

Normal execution sequence is the execution of executable statements in the order in which they appear in a program unit. The normal execution sequence begins with the first executable statement in a main program. The normal execution sequence is not affected by nonexecutable statements, or by comment lines.

A **transfer of control** is an alteration of the normal execution sequence. Statements that may be used to control the execution sequence are:

Control statement
The terminal statement of a DO loo
Input/output statements that contain an error specifier or end-of-fil specifier.

When an external procedure is referenced, execution continues with the first executable statement following the FUNCTION, SUBROUTINE, or ENTRY statement in the referenced procedure.

In this book, any description of the sequence of events in a specific transfer of control assumes that no event, such as the occurrence of an error or of a STOP statement, changes that normal sequence.

3.0 Chapter 3. Data Types and Constants

A **data type** (or **type**) is a set of values and a length. Each variable, array, constant, expression, and function has a data type.

This chapter describes:

- How type is determined
- Each type and its permitted values
- Any necessary internal representation detail (usually only the lengt of the type).
- The form of constants for each type. (Se "PARAMETER Statement" in topic 6.6 for a description of named constants.)

Subtopics

3.1 The Data Types

The nine data types are: (1)

```
INTEGER*2
INTEGER*4
REAL*4
REAL*8
COMPLEX*8
COMPLEX*16
LOGICAL*1
LOGICAL*4
CHARACTER
```

In this book, *integer* refers to INTEGER*2 or INTEGER*4, *real* refers to REAL*4 or REAL*8 (REAL*8 is the same as *double precision*), *complex* refers to COMPLEX*8 or COMPLEX*16, *logical* refers to LOGICAL*1 or LOGICAL*4, and *character* refers to CHARACTER.

(1) The form *type***length* is an abbreviation derived from the type statements. INTEGER*2, for example, has the same meaning as integer of length 2.

3.2 How Type Is Determined

Names and constants have type.

The type of a name is determined in either of two ways:

Explicitly, by a type statement (INTEGER, REAL, DOUBLE PRECISION COMPLEX, LOGICAL, or CHARACTER) or, for external functions only, by either a type statement or a FUNCTION statement.

Implicitly, by the first letter of the name

- By default (that is, in the absence of an IMPLICIT statement), if the first letter of the name is I, J, K, L, M, or N, the type is INTEGER*4. If the first letter of the name is any other letter, the type is REAL*4.
- To change, confirm, or void this default typing you may use the IMPLICIT statement.

The type of a constant is determined by the form of the constant. The discussions of types in the rest of this chapter describe the form of a constant for each type.

*3.3 INTEGER*2 Type*

Type INTEGER*2 is used for exact representations of integer values.

Length: 2 bytes.

Range of values: -32768 to 32767.

Form of constant: There are no constants (without names) of this type. Note that the PARAMETER statement may be used to specify named constants of this type.

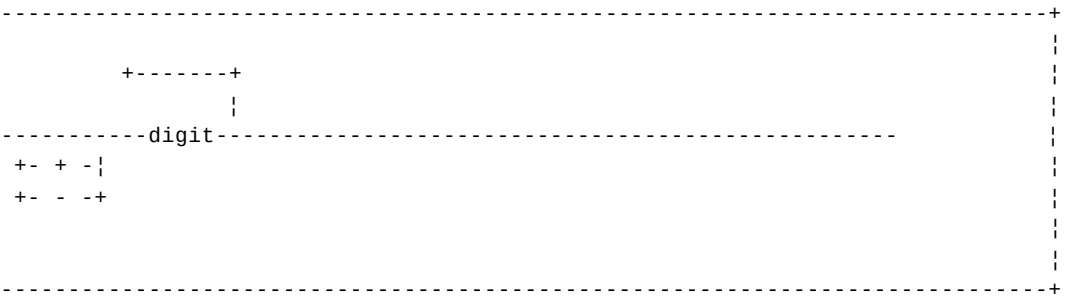
*3.4 INTEGER*4 Type*

Type INTEGER*4 is used for exact representations of integer values.

Length: 4 bytes.

Range of values: -2147483647 to 2147483647.

Form of constant:



Examples of INTEGER*4 Constants

```
25
+483
-111545
```

*3.5 REAL*4 Type*

Type REAL*4 is used for approximations of real numbers.

Length: 4 bytes.

Approximate range of values:

On MVS and VM

- 10(-78) to 10(75)
- 0
- -10(-78) to -10(75)

Precision: values have a precision of 21 to 24 bits (about six decimal digits).

¦ On OS/400 and OS/2

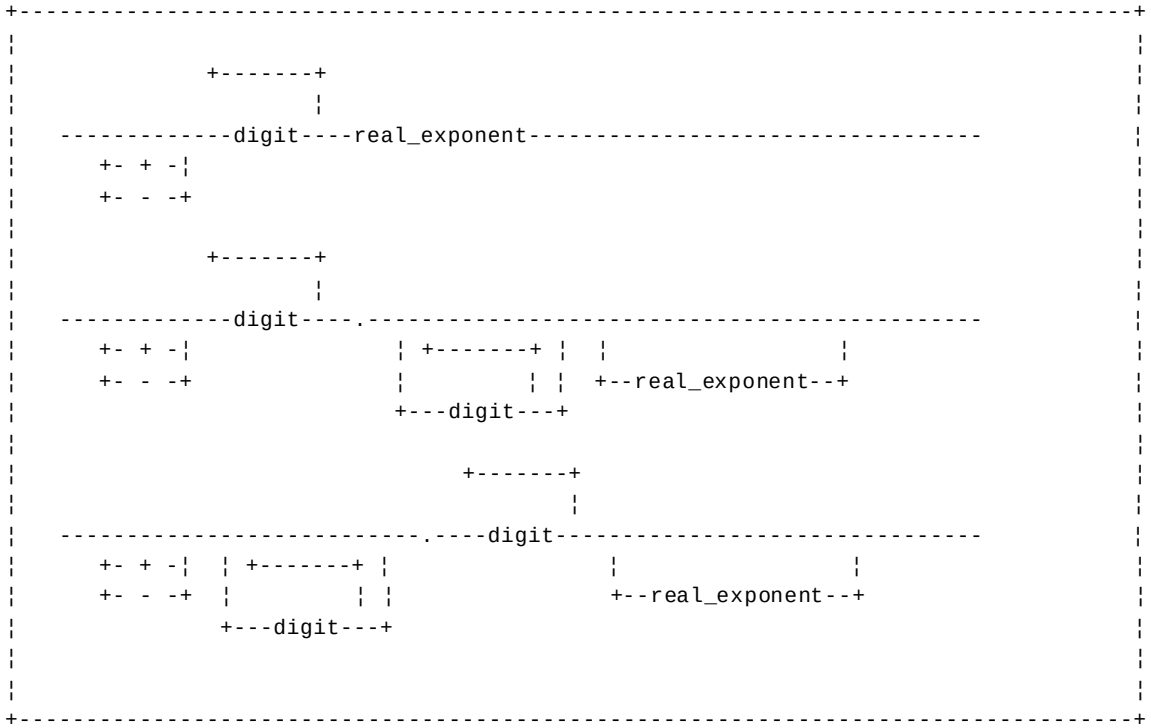
- Normalized:
  - 1.2 x 10(-38) to 3.4 x 10(38)
  - 0
  - -1.2 x 10(-38) to -3.4 x 10(38)
- Denormalized:
  - 1.4 x 10(-45) to 1.2 x 10(-38)
  - -1.4 x 10(-45) to -1.2 x 10(-38)
- NaN (not a number)
- Positive infinity
- Negative infinity.

Precision: normalized values have a precision of 24 bits (about seven decimal digits) and denormalized values may be represented with as little as one bit of precision.

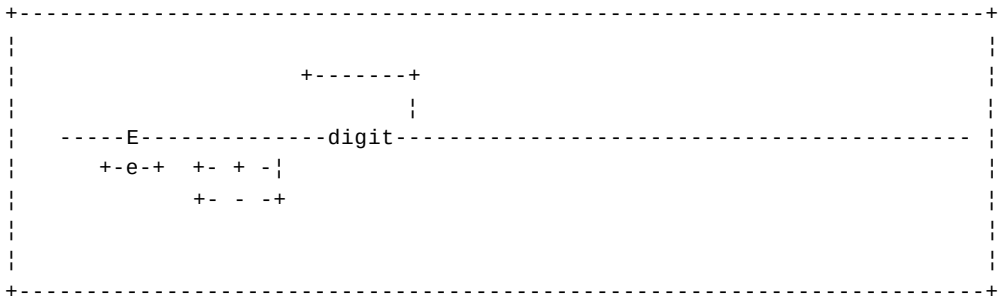
Subtopics
3.5.1 Forms of a Real Constant

3.5.1 Forms of a Real Constant

The forms of a real constant are:



The form of **real_exponent** (a **real exponent**) is:



The real exponent indicates the power of ten by which the value of the real constant without the exponent is multiplied to obtain the value of the real constant with the exponent.

The system approximates real constants, just as the real type is used for approximations of real numbers. FORTRAN products on computers with different implementations of real (floating-point) arithmetic, as in the case of VS FORTRAN, FORTRAN/400, and FORTRAN/2, have different floating-point approximations for the same real number.

Examples of Real Constants

3.14159	
-.618034	
7.0e3	That is, 7.0 x 10(3) = 7000.0

9761.25E+1	That is, 9761.25 x 10(1) = 97612.5
7e-03	That is, 7.0 x 10(-3) = 0.007
744E5	That is, 744.0 x 10(5) = 74400000.0

*3.6 REAL*8 (Double Precision) Type*

Type REAL*8, also known as type double precision, is used for approximations of real numbers.

Length: 8 bytes.

Approximate range of values:

On MVS and VM

- 10(-78) to 10(75)
- 0
- -10(-78) to -10(75)

Precision: values have a precision of 53 to 56 bits (about 16 decimal digits).

¦ On OS/400 and OS/2

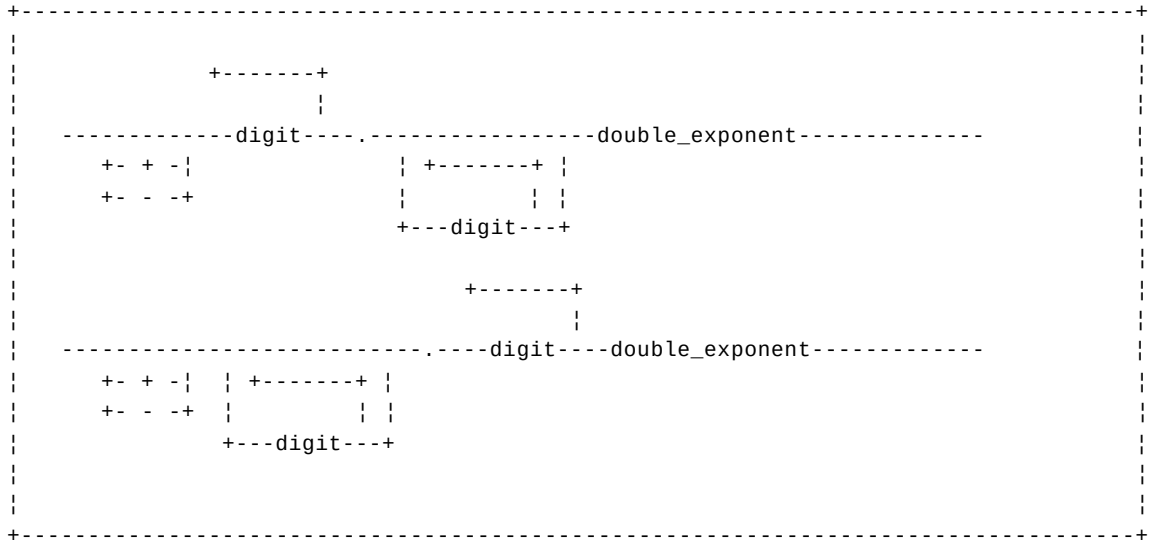
- Normalized:
  - 2.23 x 10(-308) to 1.79 x 10(308)
  - 0
  - -2.23 x 10(-308) to -1.79 x 10(308)
- Denormalized:
  - 4.94 x 10(-324) to 2.23 x 10(-308)
  - -4.94 x 10(-324) to -2.23 x 10(-308)
- NaN (not a number)
- Positive infinity
- Negative infinity.

Precision: normalized values have a precision of 52 bits (about 16 decimal digits) and denormalized values may be represented with as little as one bit of precision.

Subtopics

3.6.1 Forms of a Double Precision Constant

The forms of a double precision constant are:



The form of **double_exponent** (a **double precision exponent**) is:

The double precision exponent indicates the power of ten by which the value of the double precision constant without the exponent is multiplied to obtain the value of the double precision constant with the exponent.

The system approximates double precision constants, just as the double precision type is used for approximations of real numbers. FORTRAN products on computers with different implementations of real (floating-point) arithmetic, as in the case of VS FORTRAN, FORTRAN/400, and FORTRAN/2, have different floating-point approximations for the same real number.

Examples of Double Precision Constants

7.9D0	That is, 7.9 x 10(0) = 7.9
7.9d+03	That is, 7.9 x 10(3) = 7900.0
7D03	That is, 7.0 x 10(3) = 7000.0
20d-3	That is, 20.0 x 10(-3) = .02

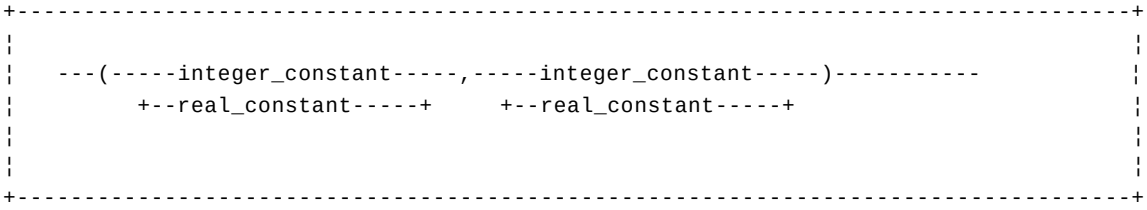
*3.7 COMPLEX*8 Type*

Type COMPLEX*8 is an approximation to the value of a complex number. The representation of a complex value is in the form of an ordered pair of real values. The first of the pair represents the real part of the complex value and the second represents the imaginary part of the complex value.

Length: 8 bytes.

Range of values: each part has the range of the REAL*4 type. See page 3.5.

Form of constant:



Examples of COMPLEX*8 Constants

```
(1, 1)
(.707, -0.707)
(-1, 2.)
(-1.5E10, 2.6e-5)
```

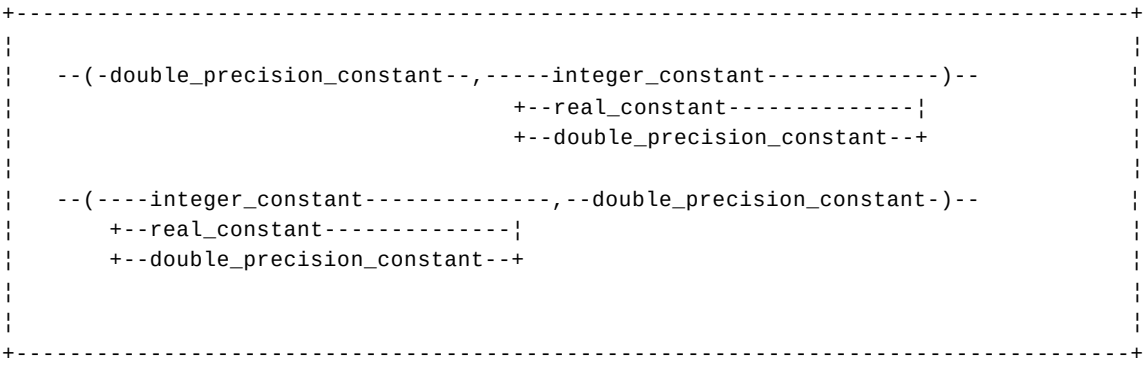
*3.8 COMPLEX*16 Type*

Type COMPLEX*16 is an approximation to the value of a complex number. The representation of a complex value is in the form of an ordered pair of double precision values. The first of the pair represents the real part of the complex value, and the second represents the imaginary part of the complex value.

Length: 16 bytes.

Range of values: each part has the range of the REAL*8 type. See page 3.6.

Forms of constant:



Examples of COMPLEX*16 Constants

```
(1, 4.4D8)
(-5.5D3, -0.446)
(3.7d10, 8.6d5)
```

*3.9 LOGICAL*1 Type*

Type LOGICAL*1 is an exact representation of the values true and false.

Length: 1 byte.

Range of values: true and false.

Form of constant: There are no constants (without names) of this type. Note that the PARAMETER statement may be used to specify named constants of this type.

*3.10 LOGICAL*4 Type*

Type LOGICAL*4 is an exact representation of the values true and false.

Length: 4 bytes.

Range of values: true and false.

Form of constant: .TRUE. (for the value true) or .FALSE. (for the value false).

3.11 CHARACTER Type

Type CHARACTER is a string of characters.

Length: 1 to 32767 bytes. The length of a character datum is the number of characters in the string. Each character in the string has a character position that is numbered consecutively 1, 2, 3, and so forth. The number indicates the sequential position of a character in the string, beginning at the left and proceeding to the right.

Range of values: The string may consist of any characters in the FORTRAN character set.

Form of constant: an apostrophe followed by a nonempty string of characters followed by an apostrophe. The delimiting apostrophes are not part of the data represented by the constant. Blanks between the delimiting apostrophes are significant. An apostrophe within the character constant is indicated by two consecutive apostrophes with no intervening blanks.

The length of a character constant is the number of characters between the delimiting apostrophes, except that each pair of consecutive apostrophes counts as a single character. The delimiting apostrophes are not counted. The length of a character constant must be greater than zero.

Examples of Character Constants

Constant	**Length**
'Systems Application Architecture FORTRAN'	40
' '	1
'The time is 1 o''clock'	21

4.0 Chapter 4. Variables, Arrays, and Character Substrings

This chapter describes:

- Variable
- Array
- Character substring
- Definition status of variables, array elements, and characte substrings
- Variable, array element, and character substring reference
- Association

Subtopics

4.1 Variables

A **variable** has a name, a type, a length, and a value that may change during program execution. The type of a variable is determined by the type of its name.

Note that an array element is not the same as a variable, as it is in some other programming languages.

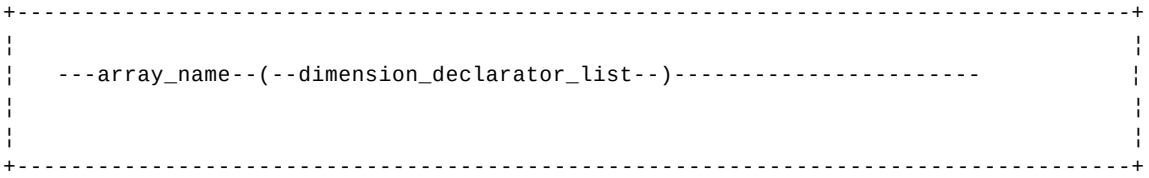
4.2 Arrays

An **array** has a name, a type, and a sequence of values. Each element of an array has an identical length and a value that may change during program execution. The type of an array is determined by the type of its name.

Subtopics

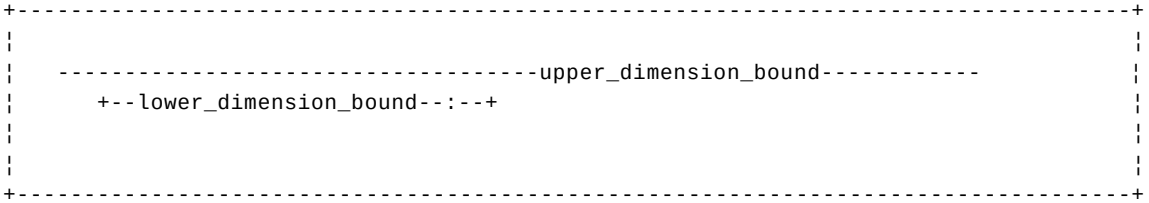
4.2.1 Array Declarators

An **array declarator** declares the name and size of an array. Every array must be declared, and no array may have more than one array declarator for the same name. An array declarator may appear in a DIMENSION, COMMON, or type statement. The form of an array declarator is:



array_name
is a name called the **array name**. Each array element has the type and length associated with this name.

dimension_declarator
A **dimension declarator** declares the lower and upper bounds of a dimension. Each dimension requires one dimension declarator. The minimum number of dimensions (and therefore dimension declarators) is one and the maximum is seven. The form of a dimension declarator is:



lower_dimension_bound
is an INTEGER*4 arithmetic expression, called a **dimension bound expression**. If this expression is not specified, a value of 1 is assumed.

upper_dimension_bound
is one of the following:

A dimension bound expression whose value must be greater than or equal to the value of the lower dimension bound
An asterisk if the dimension is the last dimension in an assumed-size array declarator.

A dimension bound expression must not contain a function or array element reference. Integer variables may appear in dimension bound expressions only in adjustable array declarators.

4.2.2 Kinds of Array Declarators and Arrays

There are three kinds of array declarators:

A **constant array declarator** is one in which every dimension bound expression is an integer constant expression.

An **adjustable array declarator** is one in which at least one of the dimension bound expressions contains at least one integer variable name. Any variable name so used must appear either in a common block or in the same dummy argument list that contains the array name. An adjustable array declarator declares an **adjustable array** and its dimensions are called **adjustable dimensions**.

An **assumed-size array declarator** is one in which the upper dimension bound of the last dimension is an asterisk.

There are two kinds of arrays:

An **actual array** is one that is declared with a constant array declarator and whose name is not a dummy argument. This kind of array may be declared in a DIMENSION statement, a COMMON statement, or a type statement.

A **dummy array** is one that may be declared with constant, adjustable, or assumed-size array declarators and whose name must be a dummy argument. This kind of array may be declared in a DIMENSION statement or a type statement.

Examples of Adjustable and Assumed-Size Array Declarators

```
subroutine SCRFCN(screen,width,lines,nops,op_codes)
integer width,lines,nops,op_codes
character*1 screen(1:width,0:lines-1)
dimension op_codes(*)
```

4.2.3 Dimensions of an Array

The size of a dimension is the value of the upper dimension bound, minus the value of the lower dimension bound, plus one. The size of a dimension that has an upper dimension bound of an asterisk is not specified.

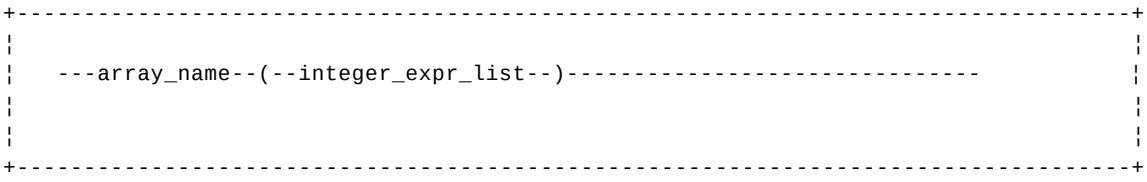
The number and size of dimensions in one array declarator may be different from the number and size of dimensions in another array declarator that is associated by common, equivalence, or argument association.

4.2.4 Size of an Array

The size of an array (that is, the number of elements in an array) is equal to the product of the sizes of its dimensions.

4.2.5 Array Elements

An array is made up of **array elements**. An array element is identified by an **array element name**, whose form is:



array_name
is a name.

integer_expr
is an integer expression called a **subscript expression**.

The number of subscript expressions must be equal to the number of dimensions in the array.

The value of each subscript expression must be greater than or equal to the corresponding lower dimension bound declared for the array. The value of each subscript expression must not exceed the corresponding upper dimension bound declared for the array. If the upper dimension bound is an asterisk, the value of the corresponding subscript expression must be such that the subscript value does not exceed the size of the actual array.

The **subscript value** determines which element of the array is identified by the array element name. The subscript value depends on the values of the subscript expressions and on the dimensions of the array. See "Arrangement of Arrays in Storage" in topic 4.2.6 for an example.

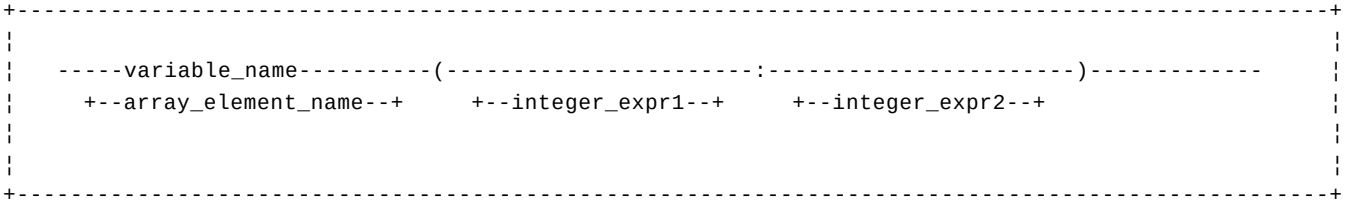
4.2.6 Arrangement of Arrays in Storage

Array elements are stored in ascending storage units in column-major order, as in the following example of a two-dimensional array declared by array declarator C(3,0:1):

	Array Element Name	**Subscript Value**
Lowest storage unit--	C(1,0)	1
	C(2,0)	2
	C(3,0)	3
	C(1,1)	4
	C(2,1)	5
Highest storage unit--	C(3,1)	6

4.3 Character Substrings

A **character substring** is a contiguous portion of a character string. A character substring is identified by a substring name, whose form is:



variable_name
is the name of a character variable.

array_element_name
is the name of a character array element.

integer_expr1 and **integer_expr2**
specify the leftmost character position and rightmost character position, respectively, of the substring. Each is an expression called a **substring expression**, which is any integer expression.

The values of **integer_expr1** and **integer_expr2** must be such that:

1 = **integer_expr1** = **integer_expr2** = **length**

where **length** is the length of the character variable or character array element. If **integer_expr1** is omitted, a value of 1 is implied. If **integer_expr2** is omitted, a value of **length** is implied.

Example of a Character Substring Name: See "Examples of Character Assignment Statements" in topic 8.4.

4.4 Definition Status

At any given time during the execution of a program, the **definition status** of each variable, array element, or character substring is either defined or undefined:

If **defined**, it has a value. The value does not change until the variable, array element, or character substring becomes undefined or until it is redefined with a different value.

If **undefined**, it does not have a predictable value.

A character variable, character array element, or character substring is defined if each of its substrings of length 1 is defined. A complex variable or complex array element is defined if each of its parts is defined.

A variable, array element, or character substring must be defined at the time its value is required. A value may be assigned (thus causing definition) by:

An assignment statement

An input statement. Each variable, array element, or characte substring in the input list becomes defined at the time it is assigned a value.

A specifier in an input/output statement

A DO statement. The DO variable becomes defined

An implied-DO list. The implied-DO variable becomes defined

A DATA statement. Initial values are provided

An ASSIGN statement

Association. Totally-associated variables of the same type, arra elements of the same type, or character substrings become defined when any one is defined. (Association is total when there is one-for-one storage mapping.)

A variable, array element, or character substring may become undefined as follows:

All are undefined at the beginning of program execution except fo those specified in DATA statements.

When a variable, array element, or character substring become defined, all associated variables, array elements, and character substrings of different type become undefined. (Complex and character types are special cases, as already described.)

An ASSIGN statement causes the specified variable to become undefine as an integer.

If a reference to a function does not need to be evaluated t determine the value of the expression in which it appears, then any variables, array elements, and character substrings in common blocks, and any arguments, that the function would have defined, become

undefined.

A RETURN or END statement causes all variables, array elements, an character substrings in the subprogram to become undefined except for the following:

- Those in a blank common block
- Those initially defined that neither were redefined nor became undefined
- Those specified by SAVE statements
- Those specified in a named common block that appears in at least one other program unit that is either directly or indirectly referencing the subprogram.

An error or end-of-file condition during an input statement causes al of the variables, array elements, and character substrings specified in the input list to become undefined.

A direct access input statement that specifies a record that was no previously written causes all of the variables, array elements, and character substrings in the input list to become undefined.

The INQUIRE statement may cause some variables, array elements, o substrings to become undefined. See "INQUIRE Statement" in topic 11.7.

4.5 Reference

A variable, array element, or character substring **reference** is the appearance of a variable name, array element name, or character substring name in a statement in a context requiring the value of the variable, array element, or character substring to be used during program execution. When a reference is executed, the current value of the variable, array element, or character substring is available. Definition of a variable, array element, or character substring is not considered a reference.

4.6 Association

Association exists if the same data item may be identified by different names in the same program unit, or by the same name or different names in different program units of the same executable program. The kinds of association are:

- Equivalence association (see page 6.2)
- Common association (see page 6.3.1)
- Entry association (see page 10.3.3)
- Argument association (see page 10.8.1).

5.0 Chapter 5. Expressions

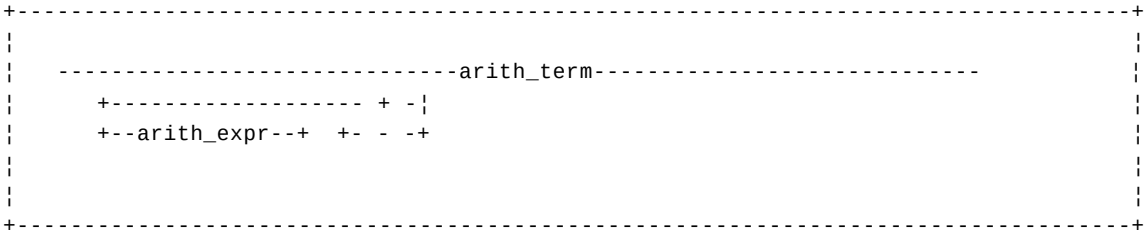
An **expression**, when evaluated, produces a value. This chapter describes the four kinds of expressions:

- Arithmetic expression
- Character expression
- Relational expression
- Logical expressions

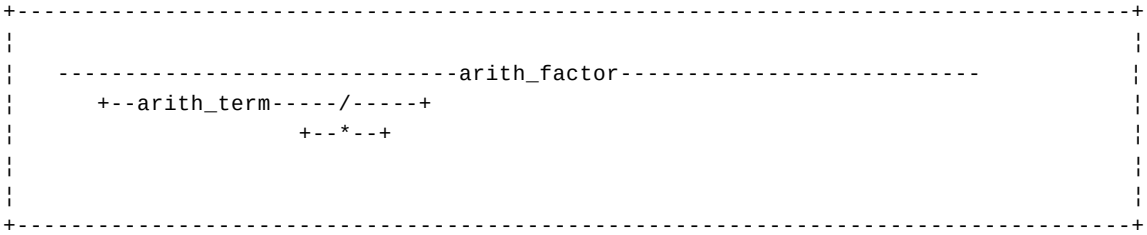
Subtopics

5.1 Arithmetic Expressions

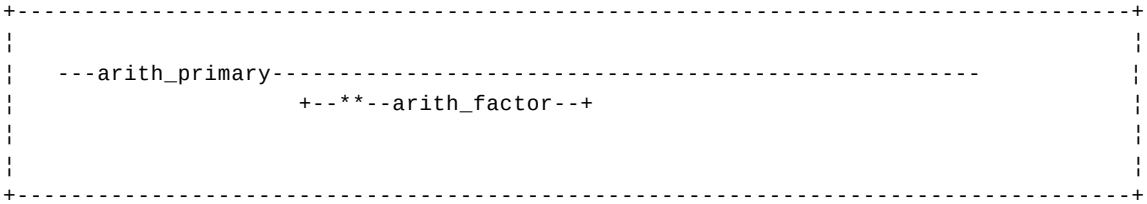
An **arithmetic expression**, when evaluated, produces a numeric value. The form of an arithmetic expression is:



The form of **arith_term** is:



The form of **arith_factor** is:



arith_primary (called a **primary**) is one of the following:

An unsigned arithmetic constan
The name of an arithmetic constan
The name of an arithmetic variabl
The name of an arithmetic array elemen
An arithmetic function referenc
An arithmetic expression enclosed in parentheses

The **arithmetic operators** are:

Arithmetic Operator	**Representing**	**Precedence**
**	Exponentiation	Highest
*	Multiplication	Intermediate
/	Division	Intermediate
+	Addition or identity	Lowest
-	Subtraction or	Lowest

	negation	

In evaluating an arithmetic expression containing two or more addition or subtraction operators, the terms are evaluated from left to right. For example, **2+3+4** is evaluated the same as **(2+3)+4**.

In evaluating a term containing two or more multiplication or division operators, the factors are evaluated from left to right. For example, **2*3*4** is evaluated the same as **(2*3)*4**.

In evaluating a factor containing two or more exponentiation operators, the primaries are combined from right to left. For example, **2**3**4** is evaluated the same as **2**(3**4)**.

In evaluating an arithmetic expression containing two or more operators having different precedence, the precedence of the operators determines the order of evaluation. For example, in the expression **-A**3**, the exponentiation operator (**) has precedence over the negation operator (-). Therefore, the operands of the exponentiation operator are combined to form an expression that is used as the operand of the negation operator. Evaluation of the expression is the same as evaluation of the expression **-(A**3)**.

Note that these formation rules do not permit expressions containing two consecutive arithmetic operators, such as **A**-B** or **A*-B**. However, expressions such as **A**(-B)** and **A*(-B)** are permitted.

Subtopics
5.1.1 Arithmetic Constant Expressions
5.1.2 Data Type of an Arithmetic Expression

5.1.1 Arithmetic Constant Expressions

An **arithmetic constant expression** is an arithmetic expression in which each primary is an arithmetic constant, the name of an arithmetic constant, or an arithmetic constant expression enclosed in parentheses. Exponentiation is permitted only if the exponent is of type integer.

An **integer constant expression** is an arithmetic constant expression in which each constant or name of a constant is of type integer.

5.1.2 Data Type of an Arithmetic Expression

Because the identity and negation operators operate on a single operand, the type of the resulting value is the same as the type of the operand.

When an arithmetic operator acts upon a pair of operands of the same type, the resulting value has that type. If the operands are of different types, the resulting value has the higher-ranking type, with the exception noted:

Rank	**Data Type**
Highest	COMPLEX*16
	COMPLEX*8 (See note.)
	REAL*8 (See note.)
	REAL*4
	INTEGER*4
Lowest	INTEGER*2

Note: If one operand is of type COMPLEX*8 and the other is of type REAL*8, the result is of type COMPLEX*16.

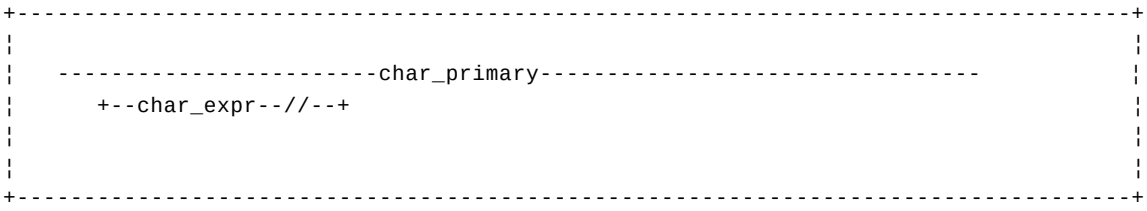
For example, addition of a COMPLEX*8 value and a REAL*4 value produces a result of type COMPLEX*8.

Examples of Arithmetic Expressions

Arithmetic Expression	**Fully Parenthesized Equivalent**
-b2/2.0**	**-((b**2)/2.0)**
ij**2**	**i**(j**2)**
a/b2 - c**	**(a/(b**2)) - c**

5.2 Character Expressions

A **character expression**, when evaluated, produces a result of type character. The form of a character expression is:



char_primary (called a **character primary**) is one of the following:

- A character constan
- The name of a character constan
- The name of a character variabl
- The name of a character array elemen
- The name of a character substrin
- A character function referenc
- A character expression enclosed in parentheses

The only **character operator** is //, representing concatenation.

In a character expression containing one or more concatenation operators, the primaries are joined, thereby forming one string whose length is equal to the sum of the lengths of the individual primaries. For example, **'AB' // 'CD' // 'EF'** is evaluated to **'ABCDEF'**. The length of the resulting string is six.

Parentheses have no effect on the value of a character expression.

Except in a character assignment statement, a character expression must not involve concatenation of an operand whose length specifier is an asterisk in parentheses (indicating inherited length) unless the operand is the name of a constant.

Subtopics
5.2.1 Character Constant Expressions

5.2.1 Character Constant Expressions

A **character constant expression** is a character expression in which each character primary is a character constant, the name of a character constant, or a character constant expression enclosed in parentheses.

Example of a Character Expression

```
      character*3 fname,lname
      data fname,lname /'Big','Ben'/
* Next line prints Big Ben
      print *,fname // ' ' // lname
```

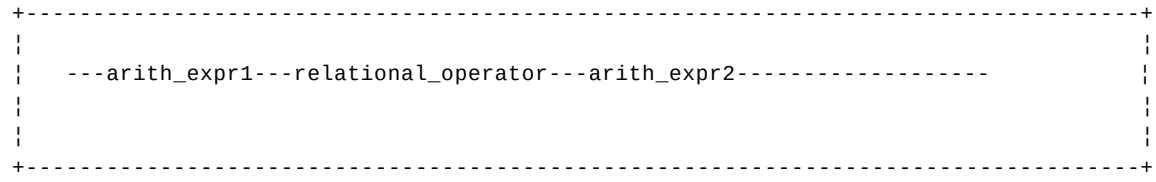
5.3 Relational Expressions

A **relational expression**, when evaluated, produces a result of type logical. A relational expression may appear only within a logical expression. A relational expression may be an arithmetic relational expression or a character relational expression.

Subtopics
5.3.1 Arithmetic Relational Expressions
5.3.2 Character Relational Expressions

5.3.1 Arithmetic Relational Expressions

An **arithmetic relational expression** compares the values of two arithmetic expressions. Its form is:



arith_expr1
arith_expr2
are each an arithmetic expression.

relational_operator
is any of the following:

Relational Operator	**Representing**
.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.GE.	Greater than or equal to

If either **arith_expr1** or **arith_expr2** is of type complex, only the relational operator .EQ. or .NE. may be specified.

If **arith_expr1** and **arith_expr2** are of different data types, the value of the relational expression is the value of the expression:

((**arith_expr1**) - (**arith_expr2**)) **relational_operator** 0

where 0 is of the same type as the expression ((**arith_expr1**) - (**arith_expr2**)).

On OS/2, if the value of either **arith_expr1** or **arith_expr2** is a NaN (not a number), the value of the relational expression is false.

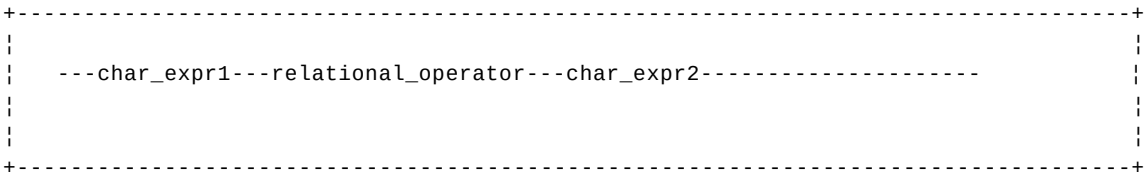
¦On OS/400, if the value of either **arith_expr1** or **arith_expr2** is a NaN (not ¦a number), the value of the relational expression is undefined.

Example of an Arithmetic Relational Expression

```
        if (e .gt. emax) emax = e
```

5.3.2 Character Relational Expressions

A **character relational expression** compares the values of two character expressions. Its form is:



char_expr1
char_expr2
are each a character expression.

relational_operator
is any of the relational operators described under "Arithmetic Relational Expressions" in topic 5.3.1.

For operators other than .EQ. and .NE., the system-dependent collating sequence (determined by the EBCDIC coded character set on MVS, VM, and OS/400, and the ASCII coded character set on OS/2) is used in interpreting a character relational expression. The character expression whose value is lower in the collating sequence, when evaluated from left to right, is considered to be less than the other. The lexical intrinsic functions (LGE, LGT, LLE, and LLT), which convert character strings to ASCII prior to comparing them, may be used to compare character strings in ASCII order.

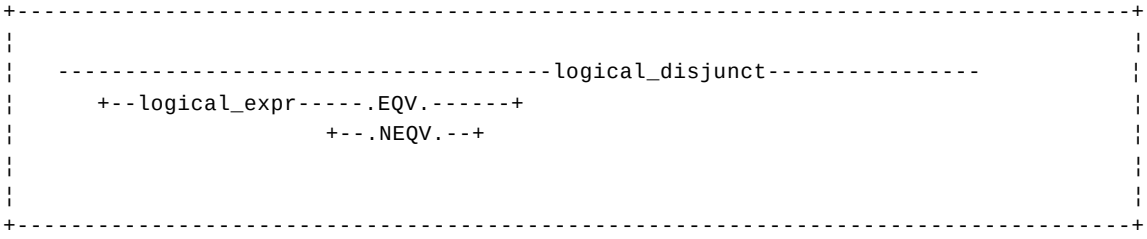
If the two character expressions are of unequal length, the shorter one is considered to be extended on the right with blanks to the length of the longer one.

Example of a Character Relational Expression

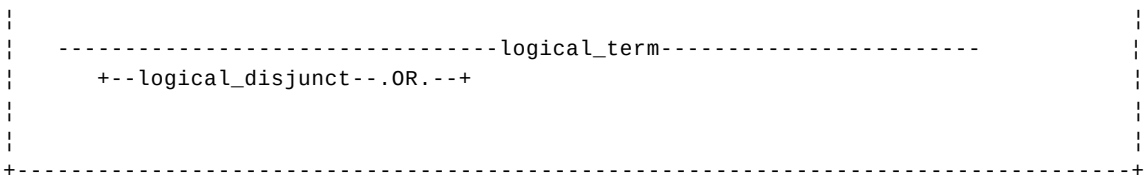
```
if (chr .gt. '0' .and. chr .le. '9') chr_type = digit
```

5.4 Logical Expressions

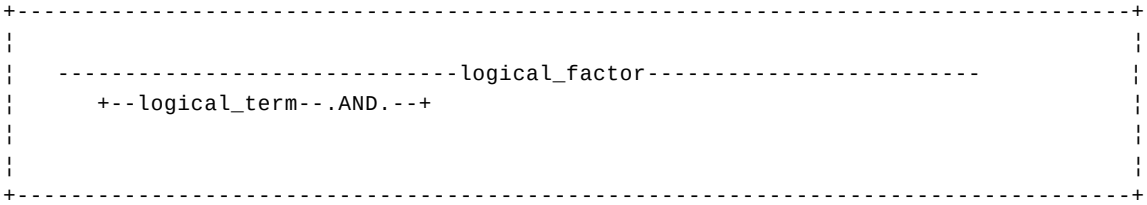
A **logical expression**, when evaluated, produces a result of type logical.
The form of a logical expression is:



The form of a **logical_disjunct** is:



The form of a **logical_term** is:



The form of a **logical_factor** is:

logical_primary (called a **logical primary**) is one of the following:

- A logical constan
- The name of a logical constan
- The name of a logical variabl
- The name of a logical array elemen
- A logical function referenc
- A relational expressio
- A logical expression enclosed in parentheses

The **logical operators** are:

Logical Operator	**Representing**	**Precedence**

.NOT.	Logical negation	Highest
.AND.	Logical conjunction	Higher
.OR.	Logical inclusive disjunction	Intermediate
.EQV.	Logical equivalence	Lowest
.NEQV.	Logical nonequivalence	Lowest

In evaluating a logical expression containing two or more operators having different precedence, the precedence of the operators determines the order of evaluation. For example, evaluation of the expression **A .OR. B .AND. C** is the same as evaluation of the expression **A .OR. (B .AND. C)**.

Subtopics
5.4.1 Value of a Logical Expression
5.4.2 Logical Constant Expressions
5.4.3 Precedence of Operators

5.4.1 Value of a Logical Expression

Assume that x[1] and x[2] represent logical values. Then, if the value of x[1] is true, the value of .NOT.x[1] is false; if the value of x[1] is false, the value of .NOT.x[1] is true. Use the following truth table to determine the values of other logical expressions:

x[1]	**x[2]**	**x[1].AND.x[2]**	**x[1].OR.x[2]**	**x[1].EQV.x[2]**	**x[1].NEQV.x[2]**
False	False	False	False	True	False
False	True	False	True	False	True
True	False	False	True	False	True
True	True	True	True	True	False

Sometimes a logical expression does not have to be completely evaluated in order to have its value determined. Consider the following logical expression (assume that LFCT is a function of type logical):

```
A .lt. B .or. LFCT(Z)
```

If A is less than B, then the function reference does not have to be evaluated to determine that this expression is true.

5.4.2 Logical Constant Expressions

A **logical constant expression** is a logical expression in which each primary is a logical constant, the name of a logical constant, a relational expression in which each primary is a constant expression, or a logical constant expression enclosed in parentheses.

5.4.3 Precedence of Operators

A logical expression may contain more than one kind of operator. When it does, the expression is evaluated from left to right, according to the following precedence among operators:

Operator	**Precedence**
Arithmetic	Highest
Character	
Relational	
Logical	Lowest

For example, the logical expression:

```
L .or. A + B .ge. C
```

where L is of type logical, and A, B, and C are of type real, is evaluated the same as the logical expression:

```
L .or. ((A + B) .ge. C)
```

6.0 Chapter 6. Specification Statements

Specification statements are nonexecutable statements that describe the characteristics and arrangement of data.

This chapter describes the specification statements:

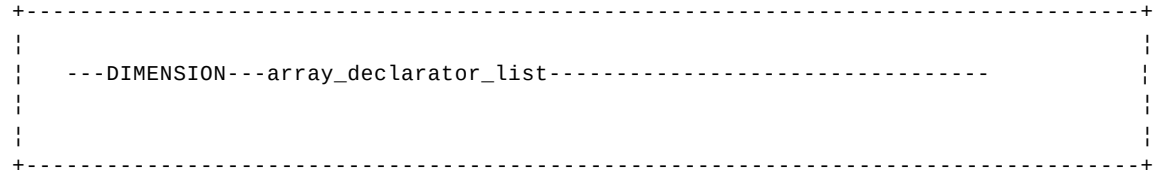
DIMENSIO
EQUIVALENC
COMMO
Type: INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, and CHARACTE
IMPLICI
PARAMETE
EXTERNA
INTRINSI
SAVE

Within a program unit, a name must not appear more than once in the same kind of specification statement, with these exceptions: a name in an EQUIVALENCE statement may appear more than once in the same or in different EQUIVALENCE statements, and a common block name may appear more than once in the same or in different COMMON statements.

Subtopics

6.1 DIMENSION Statement

MVS	VM	OS/400	OS/2
X	X	X	X

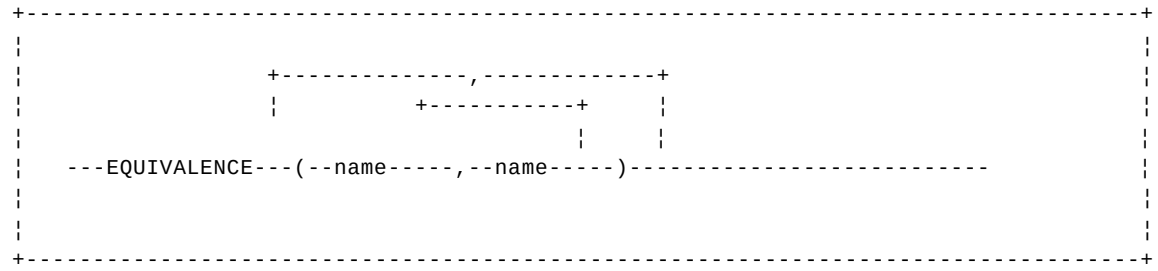


array_declarator
is an array declarator.

The DIMENSION statement specifies the names and dimension declarators of arrays. See "Arrays" starting on page 4.2 for a description of arrays and an example of the DIMENSION statement.

6.2 EQUIVALENCE Statement

MVS	VM	OS/400	OS/2
X	X	X	X



name
is one of the following:

A variable name that is not also a function name (that is, the return value of a function is not specified)
An array name
An array element name in which the subscript expressions are integer constant expressions
A character substring name in which the substring expressions are integer constant expressions.

name must not be a dummy argument name.

The EQUIVALENCE statement specifies that two or more variables, arrays, array elements, or character substrings in a program unit are to share the same storage.

All items named within a pair of parentheses have the same first storage unit and are therefore associated. This is called **equivalence association**, and it may cause the association of other items as well. (See "Example 2 of an EQUIVALENCE Statement.") Specifying an array name for **name** has the same effect as specifying an array element name that identifies the first element of the array.

Associated items may be of different data types. If so, the EQUIVALENCE statement does not cause type conversion.

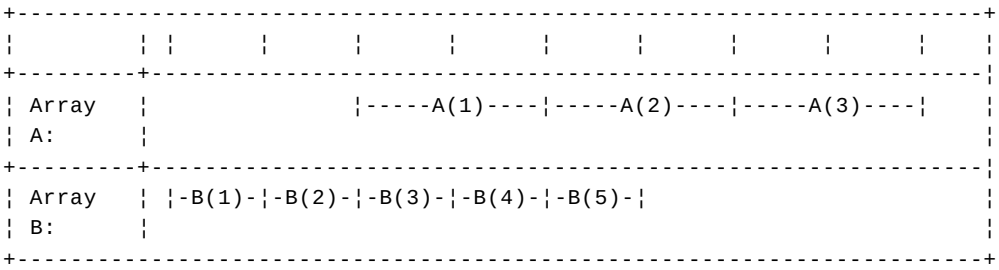
The lengths of associated items are not required to be the same.

An EQUIVALENCE statement cannot cause the storage sequences of two different common blocks to be associated.

Example 1 of an EQUIVALENCE Statement

```
double precision A(3)
real B(5)
equivalence (A,B(3))
```

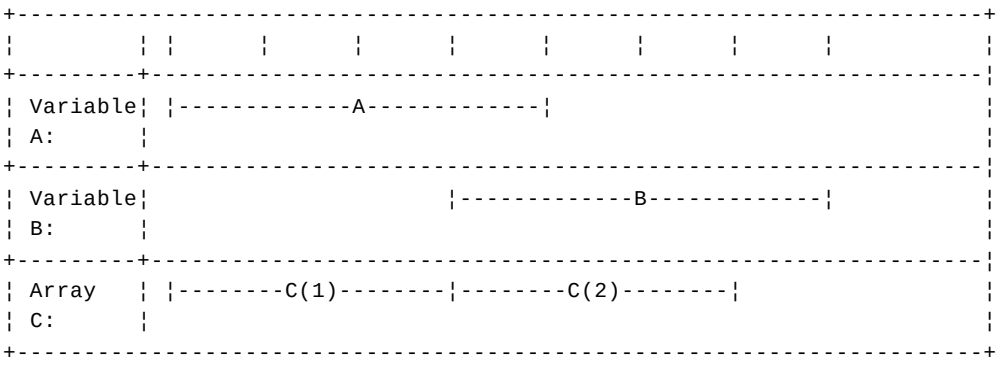
The preceding statements associate storage units as follows:



Example 2 of an EQUIVALENCE Statement: This example shows how association of two items may result in further association. The statements:

```
        character A*4,B*4,C(2)*3
        equivalence (A,C(1)),(B,C(2))
```

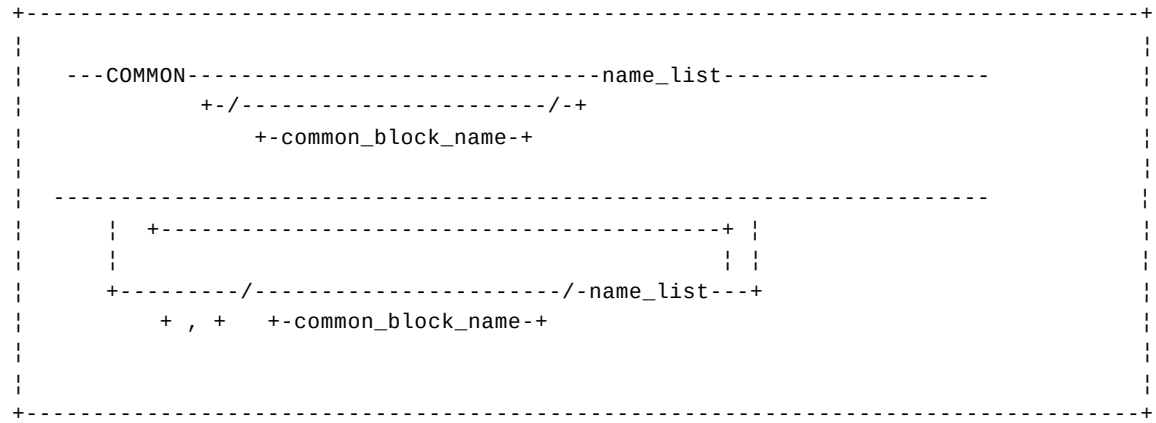
associate storage units as follows:



Because A and B are associated with C, A and B become associated with each other.

6.3 COMMON Statement

MVS	VM	OS/400	OS/2
X	X	X	X



common_block_name
is a common block name.

name
is a variable name, array name, or array declarator. None of these **names** may be used as a dummy argument name or a function name.

The COMMON statement specifies common blocks and their contents. A **common block** is a storage area that can be shared by two or more program units, allowing them to define and reference the same data and to share storage units.

A common block may be given a name. If **common_block_name** is specified, all variables and arrays specified by the following **name_list** are declared to be in that **named common block**. If **common_block_name** is omitted, all variables and arrays specified by the following **name_list** are in a **blank common block**.

Within a program unit, a common block name may appear more than once in the same or in different COMMON statements. Each successive appearance of the same common block name continues the common block specified by that name.

The variables and arrays in a common block may have different data types.

Subtopics

6.3.1 Common Association

Within an executable program, all named common blocks with the same name have the same first storage unit. Within an executable program there can be one blank common block, and all program units that refer to blank common refer to the same first storage unit. This results in the association of variables and arrays in different program units.

Because association is by storage unit, variables and arrays in a common block may have different names and types in different program units. Furthermore, a name that is used for a variable in one program unit may be used for an array in another program unit.

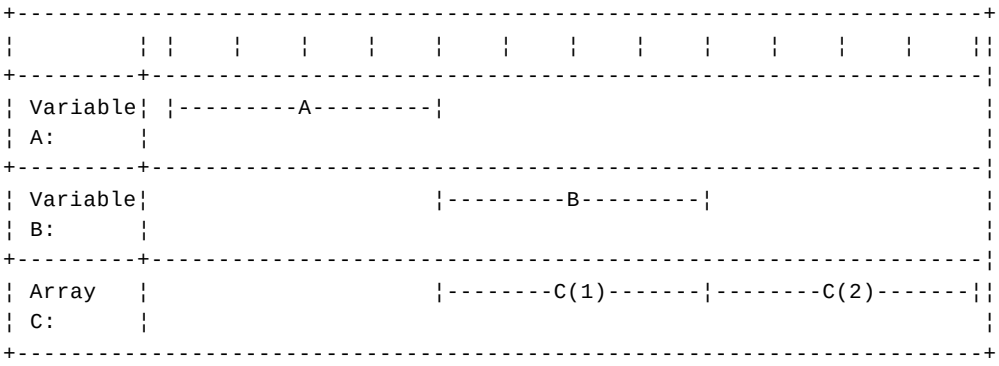
6.3.2 Common Block Storage Sequence

Variables and arrays within a common block are assigned storage units in the order that their names appear within the COMMON statement.

A common block may be extended by using an EQUIVALENCE statement, but only by adding beyond the last entry, not before the first entry. For example, the statements:

```
common /X/ A,B
real C(2)
equivalence (B,C)
```

specify common block X as follows:



6.3.3 Size of a Common Block

The size of a common block is equal to the number of bytes of storage needed to hold all the variables and arrays in the common block, including any extensions resulting from equivalence association.

6.3.4 Differences between Named Common Blocks and Blank Common Blocks

The differences between named common blocks and blank common blocks are:

Within an executable program, there may be more than one named commo block but only one blank common block.

In all program units of an executable program, named common blocks o the same name must have the same size, but blank common blocks may have different sizes. (If blank common blocks are specified with different sizes in different program units, the length of the longest becomes the length of the one blank common block in the executable program.)

Variables and array elements in a named common block may be initiall defined by using the DATA statement in a block data subprogram. Variables and array elements in a blank common block cannot be initially defined.

6.3.5 Restriction on Common and Equivalence

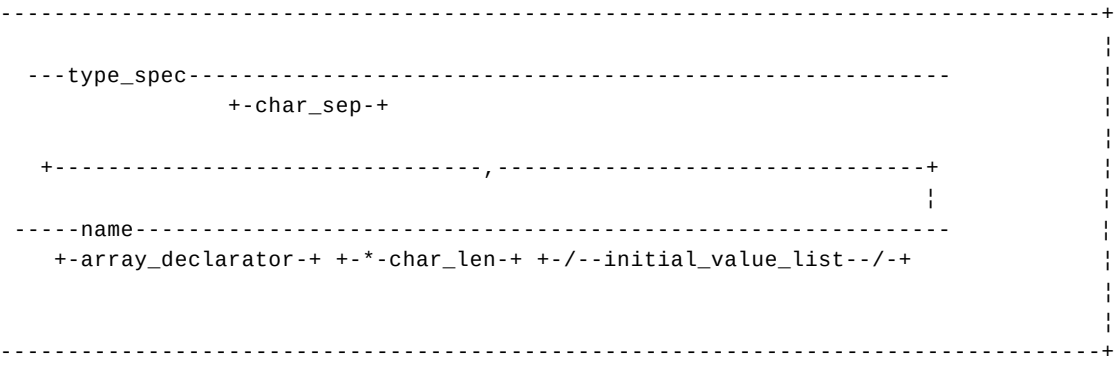
An EQUIVALENCE statement cannot cause the storage sequences of two different common blocks to become associated.

Example of a COMMON Statement

```
integer month,day,year
common /date/ month,day,year
```

6.4 INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, and CHARACTER Type Statements

MVS	VM	OS/400	OS/2
X	X	X	X



type_spec
specifies a data type and is one of the following:

INTEGER*2
INTEGER[*4]
REAL[*4]
REAL*8
DOUBLE PRECISION
COMPLEX[*8]
COMPLEX*16
LOGICAL*1
LOGICAL[*4]
CHARACTER[***char_len**]

char_sep
is a comma (,) and may be specified when **type_spec** is CHARACTER[***char_len**].

name
is a variable name, array name, name of a constant, or function name. The name must not be the name of a main program, subroutine subprogram, or block data subprogram.

array_declarator
is an array declarator.

char_len
specifies the length of items of type character. When **char_len** appears immediately after a **name** or **array_declarator**, it overrides any **char_len** specified after the word CHARACTER. **char_len** is one of the following:

An unsigned integer constant in the range 1 through 32767, inclusive.
An integer constant expression enclosed in parentheses and having the value 1 through 32767, inclusive.
An asterisk in parentheses. (See "PARAMETER Statement" in topic 6.6.)

If **char_len** is not specified, a value of 1 is assumed.

initial_value
provides an initial value for the variable or array specified by the immediately-preceding **name** or **array declarator**. This occurs just like in the DATA statement (see page 7.0).

A type statement overrides or confirms implicit typing, and may specify dimension information and initial values.

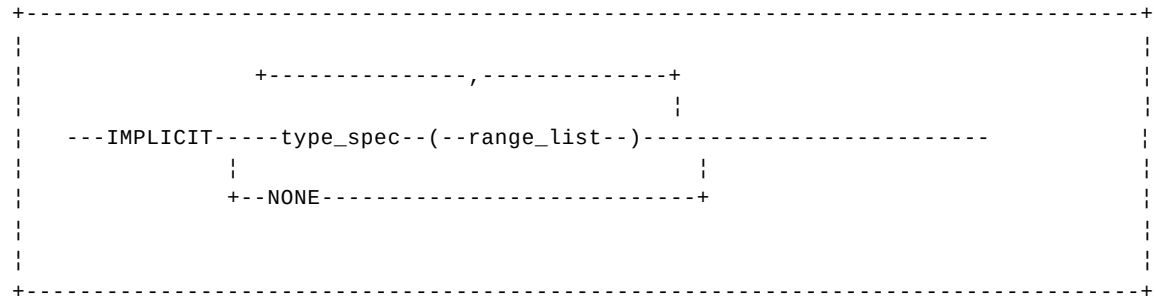
The appearance of a variable name, array name, name of a constant, external function name, or statement function in a type statement specifies the data type for that name for all appearances in the program unit. A type statement may confirm the data type of a specific intrinsic function name, but it is not required to do so. Appearance of a generic intrinsic function name is not sufficient, by itself, to remove the generic properties from the intrinsic function.

Examples of Type Statements

```
real*8 x_pos(10),y_pos(10)
logical first   / .true. /
character*(*) message_text
```

6.5 IMPLICIT Statement

MVS	VM	OS/400	OS/2
X	X	X	X



type_spec
specifies a data type and is described on page 6.4.

range
is either a single letter or a range of letters in alphabetic order. A range of letters has the form **letter[1]-letter[2]**, where **letter[1]** is the first letter in the range and **letter[2]** is the last letter in the range. Specifying a range of letters has the same effect as specifying a list of all letters in that range.

The IMPLICIT statement changes or confirms default implicit typing, or, with the form IMPLICIT NONE specified, voids implicit typing altogether. See "How Type Is Determined" in topic 3.2 for a discussion of implicit typing.

All names that begin with the letter or letters specified by **range**, and for which a type is not explicitly specified, are given the type specified by the immediately-preceding **type_spec**. The same letter may be specified only once in all the IMPLICIT statements in a program unit.

If the form IMPLICIT NONE is specified, type statements must be used to specify data types. If the form IMPLICIT NONE is specified, it must be the only IMPLICIT statement in a program unit.

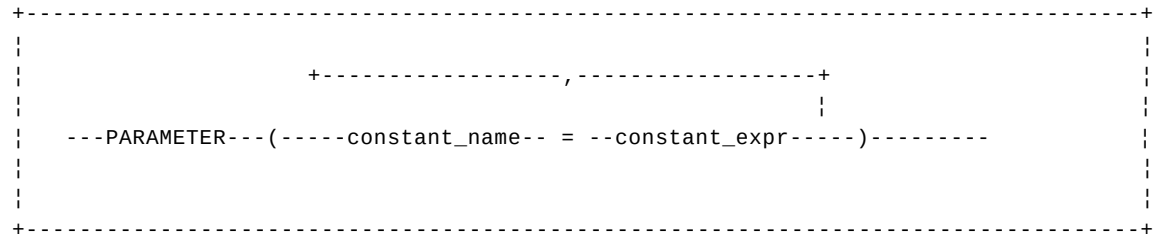
An IMPLICIT statement does not change the data type of an intrinsic function.

Example of an IMPLICIT Statement

```
        implicit integer (a), complex (q, x-z)
```

6.6 PARAMETER Statement

MVS	VM	OS/400	OS/2
X	X	X	X



constant_name
is a name called the **name of a constant**.

constant_expr
is an arithmetic constant expression, character constant expression, or logical constant expression.

The PARAMETER statement specifies names for constants. A named constant is defined with the value of **constant_expr** in accordance with the rules for assignment statements. See Chapter 8, "Assignment Statements" starting on page 8.0.

If **constant_expr** contains the name of a constant, the name must have been defined in the same or a previous PARAMETER statement.

The name of a constant must not be defined more than once in a program unit.

The name of a constant must not be part of a format specification, and it must not be used to form part of another constant, such as a complex constant.

If the data type of the name of a constant was not specified explicitly, it is determined implicitly before the constant is defined (for example, before the possible conversion of the constant expression). After the constant is defined, the data type of the name cannot be respecified.

If the name of a constant of type character has a length specifier of an asterisk in parentheses, the constant assumes (inherits) the length of its corresponding constant expression in a PARAMETER statement.

Examples of PARAMETER Statements

```
character*6 today
real pi
parameter (today = 'Friday')
parameter (pi = 3.14159)
```

6.7 EXTERNAL Statement

MVS	VM	OS/400	OS/2
X	X	X	X

name
is the name of an external procedure or block data subprogram.

The EXTERNAL statement identifies a name as an external procedure so that the name can be used as an actual argument.

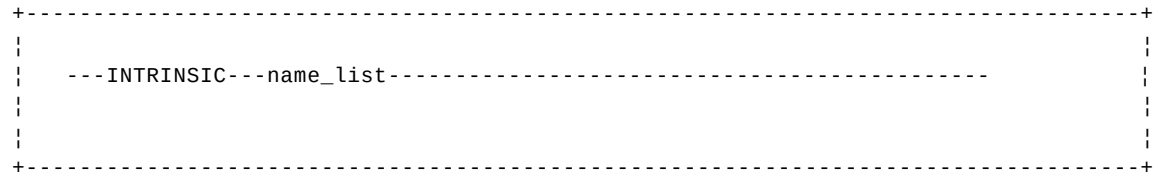
An external procedure name must appear in an EXTERNAL statement in any program unit that uses the name as an actual argument.

If an intrinsic function name appears in an EXTERNAL statement in a program unit, the name is the name of an external procedure. Therefore, an intrinsic function of the same name cannot be invoked from that program unit.

Example of an EXTERNAL Statement: See page 10.8.5.

6.8 INTRINSIC Statement

MVS	VM	OS/400	OS/2
X	X	X	X



name
is the name of an intrinsic function described in Appendix A, "Intrinsic Functions."

The INTRINSIC statement identifies a name as an intrinsic function and permits the specific names of intrinsic functions to be used as actual arguments.

If an intrinsic function name is used as an actual argument in a program unit, it must appear in an INTRINSIC statement in that program unit. The intrinsic function names that must not be used as actual arguments are those for the following:

Type conversion: INT, IFIX, IDINT, HFIX, REAL, FLOAT, SNGL, DREAL DBLE, CMPLX, DCMPLX, ICHAR, CHAR

Lexical relationships: LGE, LGT, LLE, LL

Choosing largest and smallest values: MAX, MAX0, AMAX1, DMAX1, AMAX0 MAX1, MIN, MIN0, AMIN1, DMIN1, AMIN0, MIN1

Bit-manipulation: IOR, IAND, NOT, IEOR, ISHFT, BTEST, IBSET, IBCLR

A generic function named in an INTRINSIC statement keeps its generic property.

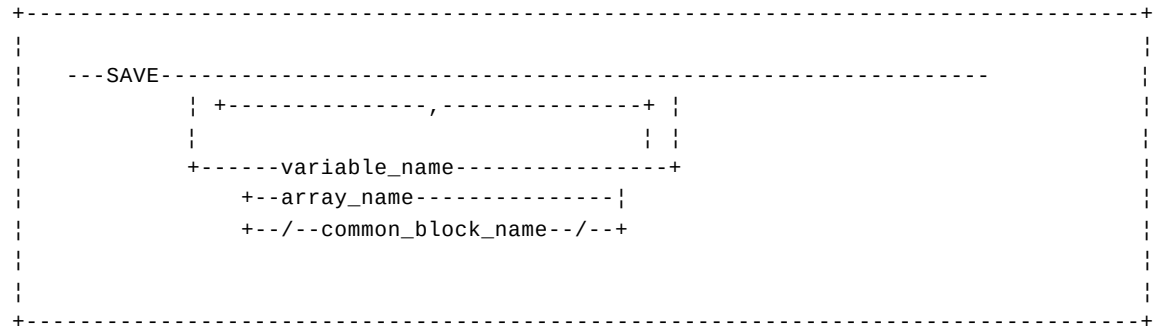
A name must not appear in both an EXTERNAL and an INTRINSIC statement in the same program unit.

Example of an INTRINSIC Statement

```
        intrinsic sin
        .
        .
  c Pass intrinsic function name to subroutine
        call doit(0.5,sin,x)
```

6.9 SAVE Statement

MVS	VM	OS/400	OS/2
X	X	X	X



variable_name
array_name
common_block_name
is the name of a variable, array, or named common block. Dummy argument names, procedure names, and the names of variables and arrays in a common block are not permitted.

The SAVE statement specifies the names of variables, arrays, and named common blocks whose definition status is to be retained after control returns from the subprogram in which the variables, arrays, and named common blocks are defined.

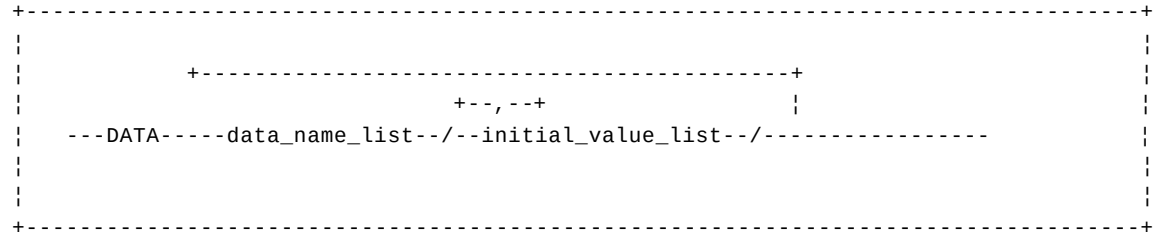
Note: In Systems Application Architecture FORTRAN, the SAVE statement is syntax-checked during compilation but has no effect during execution because the definition status of variables, arrays, and named common blocks is always retained.

A SAVE statement without a list is treated as though it contains the names of all allowable items in the program unit.

Within a function or subroutine subprogram, a variable or array whose name is specified in a SAVE statement does not become undefined as a result of the execution of a RETURN or END statement in the subprogram. However, a variable or array in a common block may become undefined or redefined in another program unit, even though the common block is specified in a SAVE statement.

7.0 Chapter 7. DATA Statement

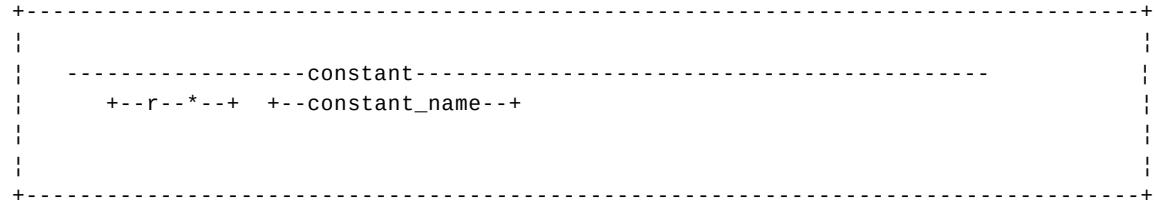
MVS	VM	OS/400	OS/2
X	X	X	X



data_name
is any of the following:

A variable name
An array name
An array element name in which the subscript expressions are integer constant expressions
A character substring name in which the substring expressions are integer constant expressions
An implied-DO list.

initial_value
has the form:



r
is a nonzero, unsigned, integer constant or the name of such a constant. The form **r*constant** or **r*constant_name** is equivalent to **r** successive appearances of the constant or name of constant.

The DATA statement is a nonexecutable statement that provides initial values for variables, array elements, and character substrings.

Each **data_name_list** must specify the same number of items as its corresponding **initial_value_list**. There is a one-to-one correspondence between the items in these two lists such that the first **data_name** corresponds to the first **initial_value**, the second **data_name** corresponds to the second **initial_value**, and so forth. This correspondence establishes the initial value of each **data_name**.

Specifying an array name as a **data_name** is the same as specifying a list of all the elements in the array in the order they are stored.

The data type of each **data_name** and the data type of its corresponding **initial_value** must agree if either is of type logical or character. For

type character, definition proceeds according to the rules for character assignment statements. (See "Character Assignment Statement" in topic 8.4.) If a **data_name** is of type integer, real, or complex, its corresponding **initial_value** need not agree in type, although it must be one of the types in that group (integer, real, or complex); if the types do not agree, the **initial_value** is converted.

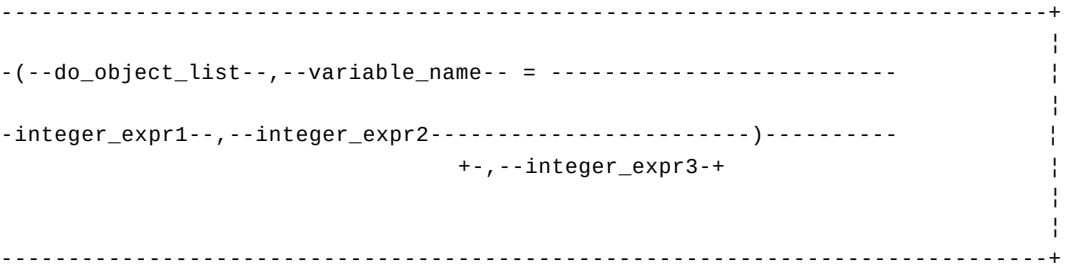
To provide an initial value for a variable, array element, or character substring in a named common block, the DATA statement must be in a block data subprogram.

A DATA statement cannot provide an initial value for:

A dummy argumen
A variable, array element, or character substring in a blank commo block, including a variable, array element, or character substring that is associated with a variable, array element, or character substring in a blank common block
A variable in a function subprogram whose name is the same as that o the function subprogram or an entry in the function subprogram.

A variable, array element, or character substring must not be initialized more than once in an executable program. If two or more variables, array elements, or character substrings are associated, only one may be initialized in a DATA statement.

An implied-DO list may be used in a DATA statement to initialize the elements of an array. Its form is:



do_object
is an array element name, character substring name, or implied-DO list.

variable_name
is the name of an INTEGER*4 variable called the **implied-DO variable**.

integer_expr1
integer_expr2
integer_expr3
are each an INTEGER*4 constant expression, except that the expression may contain implied-DO variables of other implied-DO lists that have this implied-DO list within their ranges.

The range of an implied-DO list is the list **do_object_list**. The iteration count and the values of the implied-DO variable are established from **integer_expr1**, **integer_expr2**, and **integer_expr3**, the same as for a DO statement, except that the iteration count must be positive. (See "Execution of a DO Statement" in topic 9.7.3.) When the implied-DO list is executed, the items in the **do_object_list** are specified once for each iteration of the implied-DO list, with the appropriate substitution of

values for any occurrence of the implied-DO variable.

Each subscript expression in the **do_object_list** must be an integer constant expression, except that the expression may contain implied-DO variables of implied-DO lists that have the subscript expression within their ranges.

Examples of DATA Statements

```
       integer z(100),even_odd(0:9)
       logical first_time
       data    first_time / .true. /
       data    z  / 100* 0 /
 C Implied-DO list
       data  (even_odd(j),j=0,8,2) / 5 * 0 /
     +      ,(even_odd(j),j=1,9,2) / 5 * 1 /
```

8.0 Chapter 8. Assignment Statements

Assignment statements are executable statements that assign values to variables, array elements, or character substrings.

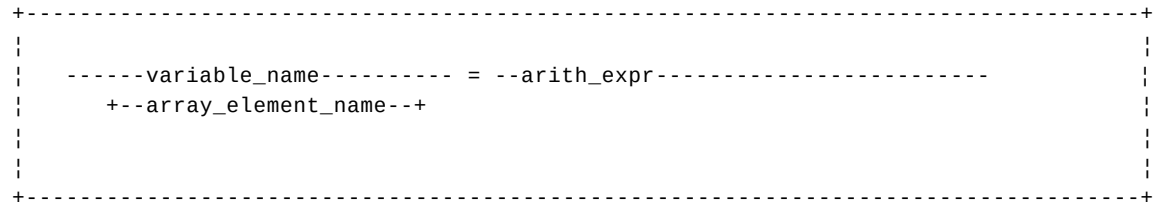
This chapter describes the four kinds of assignment statements:

- Arithmetic assignment statement
- Logical assignment statement
- Statement label (ASSIGN) assignment statement
- Character assignment statements

Subtopics

8.1 Arithmetic Assignment Statement

MVS	VM	OS/400	OS/2
X	X	X	X



variable_name
array_element_name
is the name of a variable or array element of type integer, real, or complex.

arith_expr
is an arithmetic expression.

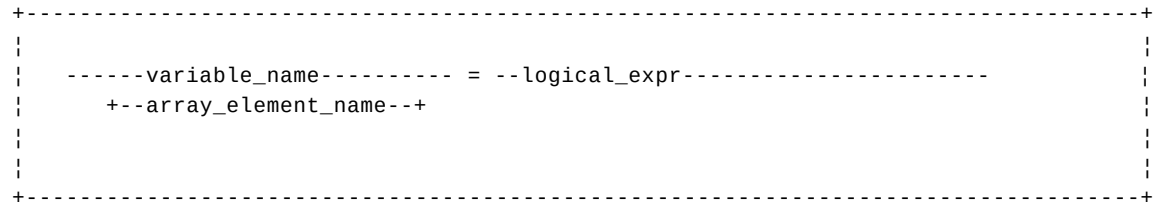
The arithmetic assignment statement evaluates **arith_expr**, converts the resulting value to the type of the variable or the array element, and assigns that value to the variable or array element.

Examples of Arithmetic Assignment Statements

```
esq = p**2 + m**2
root(1) = (-b + sqrt(b**2 - 4.0*a*c))/(2.0*a)
```

8.2 Logical Assignment Statement

MVS	VM	OS/400	OS/2
X	X	X	X



variable_name
array_element_name
is the name of a variable or array element of type logical.

logical_expr
is a logical expression.

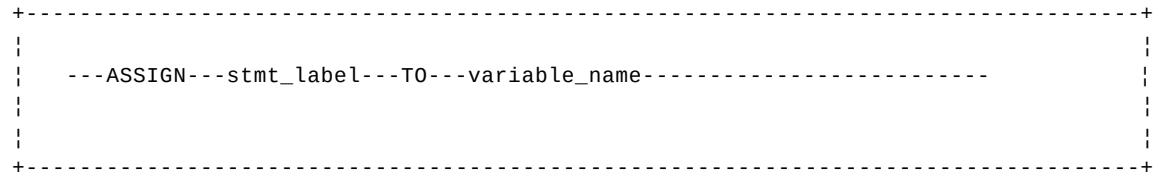
The logical assignment statement evaluates **logical_expr** and assigns the resulting value to the variable or the array element.

Example of a Logical Assignment Statement

```
logical inside
.
.
real    r,rmin,rmax
inside = (r .ge. rmin) .and. (r .le. rmax)
```

8.3 Statement Label Assignment (ASSIGN) Statement

MVS	VM	OS/400	OS/2
X	X	X	X



stmt_label
specifies the statement label of an executable statement or a FORMAT statement.

variable_name
is the name of an INTEGER*4 variable.

The ASSIGN statement assigns a statement label to the variable.

A variable must be defined with a statement label value when referenced in an assigned GO TO statement or as a format identifier in an input/output statement. While defined with a statement label value, the variable must not be referenced in any other way.

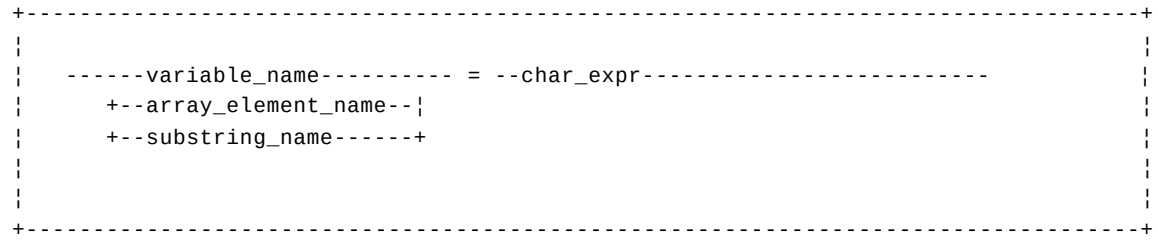
Use of an ASSIGN statement is the only way to define a variable with a statement label value.

An integer variable defined with a statement label value may be redefined by another ASSIGN statement or with an integer value.

Example of a Statement Label Assignment (ASSIGN) Statement: See "Example of an Assigned GO TO Statement" in topic 9.3.

8.4 Character Assignment Statement

MVS	VM	OS/400	OS/2
X	X	X	X



variable_name
array_element_name
substring_name
is the name of a character variable, character array element, or character substring.

char_expr
is a character expression.

The character assignment statement evaluates **char_expr** and assigns the resulting value to the character variable, character array element, or character substring.

None of the character positions being defined in the character variable, character array element, or character substring may be referenced by the character expression.

If the length of the character variable, character array element, or character substring is greater than the length of the character expression, the character expression is extended to the right with blanks until the lengths are equal, and then assigned. If the length of the character variable, character array element, or character substring is less, the character expression is truncated on the right to match the length of the character variable, character array element, or character substring, and then assigned.

Only as much of the character expression need be defined as is necessary to define the character variable, character array element, or character substring. For example:

```
character Scott*4, Dick*8
Scott = Dick
```

This assignment of **Dick** to **Scott** requires that the substring **Dick(1:4)** be defined. The rest of **Dick**, **Dick(5:8)**, does not have to be defined.

If **substring_name** is specified, the character expression is assigned only to the character substring identified by that **substring_name**. The definition status of other character substrings does not change.

Examples of Character Assignment Statements

```
character*80 line, ch*1, seq*8
.
.
ch = line(1:1)
seq = line(73:80)
```

9.0 Chapter 9. Control Statements

Control statements are executable statements that control the execution sequence of a program.

This chapter describes all the control statements except for CALL and RETURN (which are described in Chapter 10, "Program Units and Procedures"). The control statements described in this chapter are:

- Unconditional GO T
- Computed GO T
- Assigned GO T
- Arithmetic I
- Logical I
- Block IF, ELSE IF, ELSE, and END IF (grouped in an IF construct
- D
- CONTINU
- STO
- PAUS
- END

Subtopics

9.1 Unconditional GO TO Statement

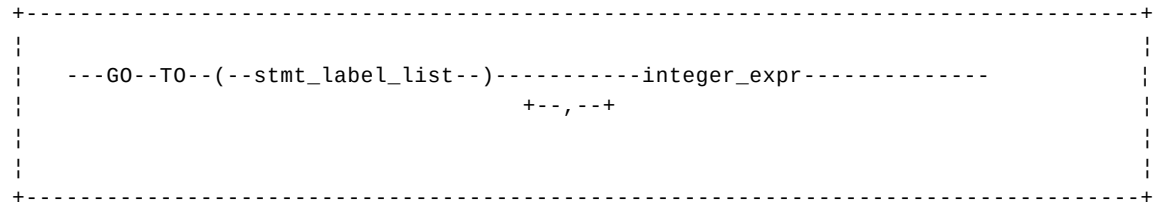
MVS	VM	OS/400	OS/2
X	X	X	X

stmt_label
is the statement label of an executable statement.

The unconditional GO TO statement transfers control to the statement identified by **stmt_label**.

9.2 Computed GO TO Statement

MVS	VM	OS/400	OS/2
X	X	X	X



stmt_label
is the statement label of an executable statement. The same statement label may appear more than once in **stmt_label_list**.

integer_expr
is an integer expression.

The computed GO TO statement evaluates **integer_expr**, uses the resulting value as an index into **stmt_label_list**, and transfers control to the statement whose statement label is identified by that index. For example, if the value of **integer_expr** is 4, control transfers to the statement whose statement label is fourth in the **stmt_label_list**.

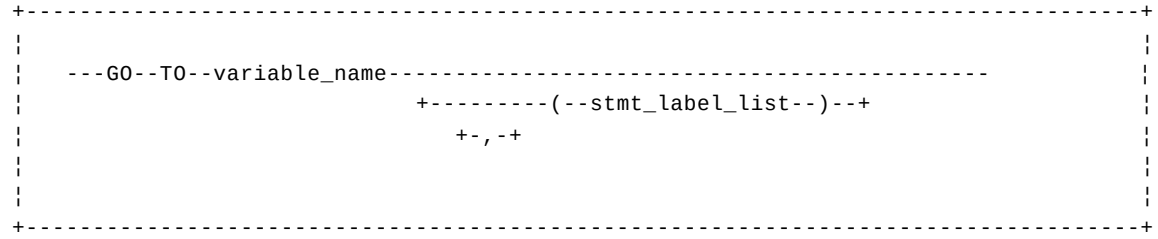
If the value of **integer_expr** is less than one or greater than the number of statement labels in the list, execution of the GO TO statement has no effect (like a CONTINUE statement).

Example of a Computed GO TO Statement

```
        integer next
        .
        .
        go to (100,200) next
  10    print *,'Execution transfers here if NEXT does not equal 1 or 2'
        .
        .
  100   print *,'Execution transfers here if NEXT = 1'
        .
        .
  200   print *,'Execution transfers here if NEXT = 2'
```

9.3 Assigned GO TO Statement

MVS	VM	OS/400	OS/2
X	X	X	X



variable_name
is a variable name of type INTEGER*4.

stmt_label
is the statement label of an executable statement. The same statement label may appear more than once in **stmt_label_list**.

The assigned GO TO statement transfers control to the statement identified by a statement label. At the time the assigned GO TO statement is executed, the variable specified by **variable_name** must be defined with the value of a statement label. This definition must be made with an ASSIGN statement in the same program unit as the assigned GO TO statement.

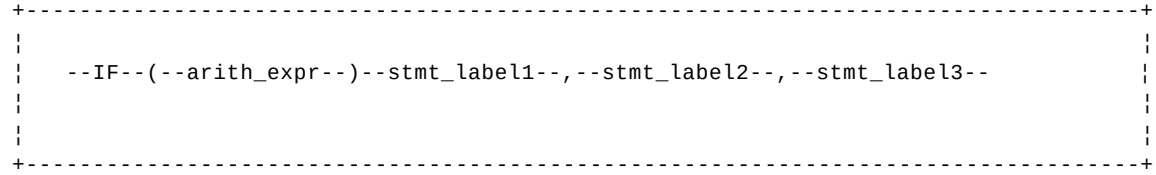
If **stmt_label_list** is present, the statement label assigned to the variable specified by **variable_name** must be one of the statement labels in the list.

Example of an Assigned GO TO Statement

```
        integer return_label
        .
        .
  c  Assign the return label and "call" the local procedure
        assign 100 to return_label
        goto 9000
  100   continue
        .
        .
  9000  continue
  c A "local" procedure.
        .
        .
        goto return_label
```

9.4 Arithmetic IF Statement

MVS	VM	OS/400	OS/2
X	X	X	X



arith_expr
is an integer or real expression.

stmt_label1
stmt_label2
stmt_label3
are statement labels of executable statements. The same statement label may appear more than once among the three statement labels.

The arithmetic IF statement evaluates **arith_expr** and transfers control to the statement identified by **stmt_label1**, **stmt_label2**, or **stmt_label3**, depending on whether the value of **arith_expr** is less than zero, zero, or greater than zero, respectively.

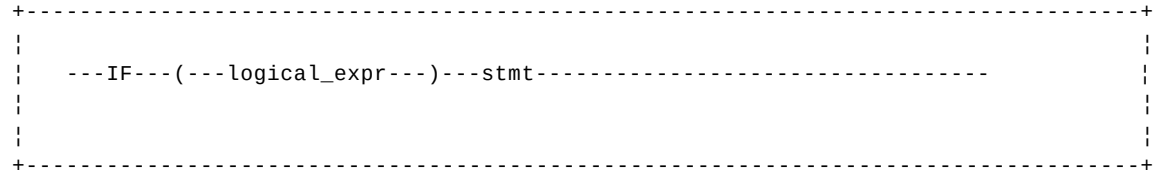
¦On OS/400 and OS/2, the value of **arith_expr** must not be a NaN (not a ¦number).

Example of an Arithmetic IF Statement

```
      if (k-100) 10,20,30
10    print *,'K is less than 100.'
      go to 40
20    print *,'K equals 100.'
      go to 40
30    print *,'K is greater than 100.'
40    continue
```

9.5 Logical IF Statement

MVS	VM	OS/400	OS/2
X	X	X	X



logical_expr
is a logical expression.

stmt
is any unlabeled executable statement except DO, block IF, ELSE IF, ELSE, END IF, END, or another logical IF.

The logical IF statement evaluates **logical_expr** and uses the resulting value to determine whether **stmt** is processed. If the value of **logical_expr** is true, **stmt** is executed. If the value of **logical_expr** is false, **stmt** is not executed and the IF statement has no effect (like a CONTINUE statement).

Execution of a function reference in **logical_expr** may change the values of data items in **stmt**.

Example of a Logical IF Statement

```
        if (err.ne.0) call error(err)
```

9.6 IF Construct--Block IF, ELSE IF, ELSE, and END IF Statements

MVS	VM	OS/400	OS/2
X	X	X	X

```
IF (logical_expr) THEN

      [stmt_block]

  [ELSE IF (logical_expr) THEN

      [stmt_block]]...

  [ELSE

      [stmt_block]]

END IF
```

logical_expr
is a logical expression.

stmt_block
is a statement block consisting of zero or more executable statements.

The **IF construct** controls the execution sequence. It is made up of a block IF statement, an END IF statement, and, optionally, ELSE IF, ELSE, and other executable statements. The box above shows the statements in an IF construct in their required sequence.

The logical expressions in an IF construct are evaluated in the order of their appearance until a true value, an ELSE statement, or an END IF statement is found:

If a true value or an ELSE statement is found, the statement bloc immediately following is executed, and the execution of the IF construct is complete. The logical expressions in any remaining ELSE IF statements of the IF construct are not evaluated.

If an END IF statement is found, no statement blocks execute, and th execution of the IF construct is complete.

Transfer of control into an IF construct from outside it is not permitted. Transfer of control within an IF construct is permitted within statement blocks, but is not permitted between statement blocks or to an ELSE IF or ELSE statement.

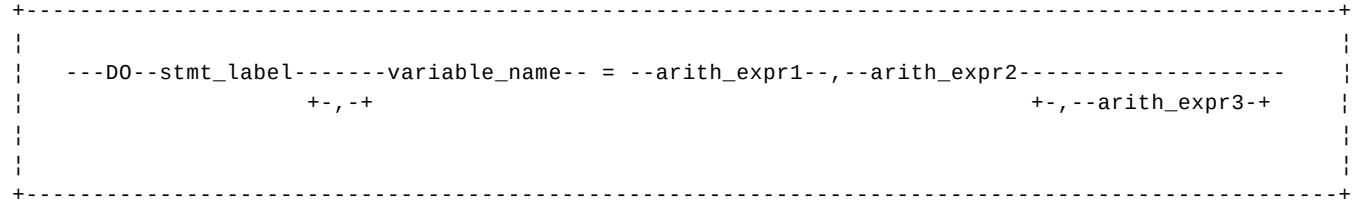
IF constructs may be nested, that is, any of the statement blocks may contain IF constructs.

Example of an IF Construct

```
c Get a record (containing a command) from the terminal
100   continue
      .
      .
c Process the command
      if (cmd .eq. 'retry') then
           if (limit .gt. five) then
c               Print retry limit exceeded
                .
                .
                call stop
           else
                call retry
           end if
      else if (cmd .eq. 'stop') then
           call stop
      else if (cmd .eq. 'abort') then
           call abort
      else
           go to 100
      end if
```

9.7 DO Statement

MVS	VM	OS/400	OS/2
X	X	X	X



stmt_label
    is the statement label of the **terminal statement**, which is the executable statement at the end of the DO loop.

variable_name
    is the name of an integer or real variable called the **DO variable**.

arith_expr1
arith_expr2
arith_expr3
    are each an integer or real expression.

The DO statement specifies a loop, called a **DO loop**.

The terminal statement must follow the DO statement and must not be any of the following statements: unconditional GO TO, assigned GO TO, arithmetic IF, block IF, ELSE IF, ELSE, END IF, RETURN, STOP, END, or DO.

Subtopics

9.7.1 Range of a DO Loop

The **range of a DO loop** consists of all the executable statements following the DO statement, up to and including the terminal statement. Concerning the range:

If a DO statement appears within the range of a DO loop (that is, i nested), the range of the nested DO loop must be entirely within the range of the outer DO loop.

DO loops may share a terminal statement

If a DO statement appears within a statement block of an IF construct the range of the DO loop must be contained entirely within that statement block.

If an IF construct appears within the range of a DO loop, no part o the construct may appear outside the range.

Transfer of control into the range of a DO loop from outside the rang is not permitted.

Transfer of control to a shared terminal statement may only be don from the innermost sharing DO loop.

9.7.2 Active and Inactive DO Loops

A DO loop is either active or inactive. Initially inactive, a DO loop becomes active only when its DO statement is executed. Once active, the DO loop becomes inactive only when:

Its iteration count becomes zero

A RETURN statement is executed within the range of the DO loop

Control is transferred to a statement in the same program unit bu outside the range of the DO loop.

A subroutine invoked from within the DO loop returns, via an alternat return specifier, to a statement that is outside the range of the DO loop.

A STOP statement is executed or execution is terminated for any othe reason.

When a DO loop becomes inactive, the DO variable keeps the last value assigned to it.

9.7.3 Execution of a DO Statement

1. The initial parameter, **m[1]**, the terminal parameter, **m[2]**, and the incrementation parameter, **m[3]** are established by evaluating **arith_expr1**, **arith_expr2**, and **arith_expr3**, respectively. Evaluation includes, if necessary, conversion to the type of the DO variable. If **arith_expr3** is not specified, **m[3]** has a value of 1. **m[3]** must not have a value of zero.

2. The DO variable becomes defined with the value of the initial parameter (**m[1]**).

3. The **iteration count** is established and is the value of the expression:

   MAX (INT ((**m[2]** - **m[1]** + **m[3]**) / **m[3]**), 0)

   Note that the iteration count is 0 whenever:

   m[1] > **m[2]** and **m[3]** > 0, or
   m[1] < **m[2]** and **m[3]** < 0

At the completion of execution of the DO statement, loop control processing begins.

9.7.4 Loop Control Processing

Loop control processing determines if further execution of the range of the DO loop is required. The iteration count is tested. If the count is not zero, execution of the first statement in the range of the DO loop begins. If the iteration count is zero, the DO loop becomes inactive. If, as a result, all of the DO loops sharing the terminal statement of this DO loop are inactive, normal execution continues with the execution of the next executable statement following the terminal statement. However, if some of the DO loops sharing the terminal statement are active, execution continues with incrementation processing.

9.7.5 Execution of the Range

Statements in the range of the DO loop are executed until reaching the terminal statement. Except by incrementation processing, the DO variable may neither be redefined nor become undefined during execution of the range of the DO loop.

9.7.6 Terminal Statement Execution

Execution of the terminal statement occurs as a result of the normal execution sequence, or as a result of transfer of control, subject to the restriction that transfer of control into the range of a DO loop from outside the range is not permitted. Unless execution of the terminal statement results in a transfer of control, execution then continues with incrementation processing.

9.7.7 Incrementation Processing

1. The DO variable, the iteration count, and the incrementation parameter (**m[3]**) of the active DO loop whose DO statement was most recently executed, are selected for processing.

2. The value of the DO variable is incremented by the value of **m[3]**.

3. The iteration count is decremented by 1.

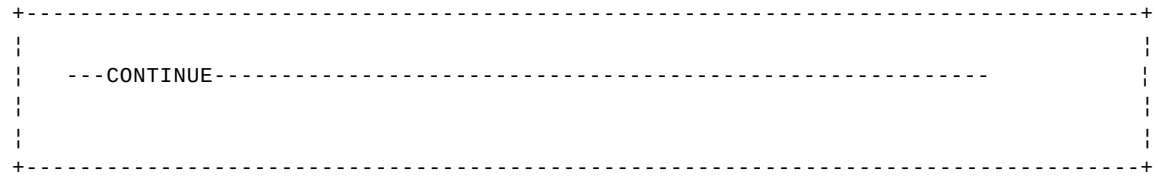
4. Execution continues with loop control processing of the same DO loop whose iteration count was decremented.

Examples of DO Statements

```
      do 20 i = 2, 5
         earliest(i) = 0.0
         do 10 j = 1, i-1
            if (network(j,i) .ne. 0.0)
     x         earliest(i) = max(network(j,i)+earliest(j), earliest(i))
10       continue
20    continue
```

9.8 CONTINUE Statement

MVS	VM	OS/400	OS/2
X	X	X	X



The CONTINUE statement has no effect.

9.9 STOP Statement

MVS	VM	OS/400	OS/2
X	X	X	X

char_constant
is a character constant of 1 through 72 characters.

digit_string
is a string of 1 through 5 digits.

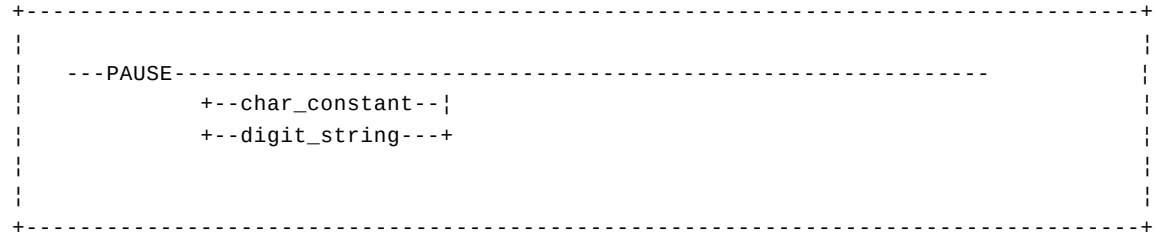
The STOP statement stops the execution of a program and displays **char_constant** or **digit_string**, if specified, to the user.

Examples of STOP Statements

```
STOP 'Abnormal Termination'
STOP 15
```

9.10 PAUSE Statement

MVS	VM	OS/400	OS/2
X	X	X	X



char_constant
is a character constant of 1 through 72 characters.

digit_string
is a string of 1 through 5 digits.

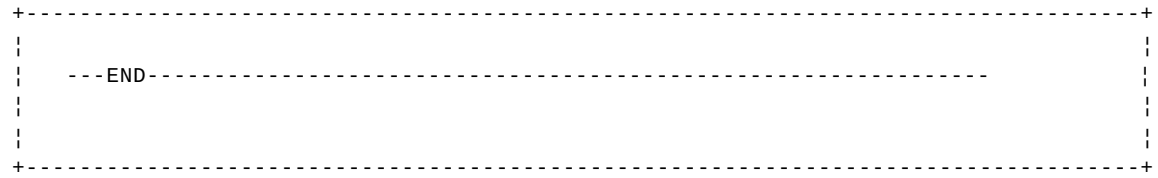
The PAUSE statement suspends the execution of a program and displays **char_constant** or **digit_string**, if specified, to the user. When the user intervenes, execution resumes as though a CONTINUE statement were executed.

Examples of PAUSE Statements

```
PAUSE 'Insert a diskette into the default drive.'
PAUSE 10
```

9.11 END Statement

MVS	VM	OS/400	OS/2
X	X	X	X



The END statement is the final statement in a program unit. It is the only required statement.

The END statement in a main program terminates execution of the executable program. The END statement in a function or subroutine subprogram has the same effect as a RETURN statement.

10.0 Chapter 10. Program Units and Procedures

This chapter describes:

- Relationships among program units and procedure
- Functions and subroutine
- Argument
- The PROGRAM, FUNCTION, statement function, SUBROUTINE, CALL, ENTRY RETURN, and BLOCK DATA statements.

Subtopics

10.1 Relationships among Program Units and Procedures

A **program unit** is a sequence of statements and optional comment lines, with the final statement being an END statement. An **executable program** is a collection of program units consisting of one main program and zero or more subprograms.

Program unit relationships are illustrated in Figure 1.

A **procedure** may be invoked by a program unit to perform a particular activity. When a **procedure reference** is made, the referenced procedure is executed.

Procedure relationships are illustrated in Figure 2.

Recursion is not permitted. That is, a program unit must not invoke itself, either directly or indirectly (via a program unit that it invokes).

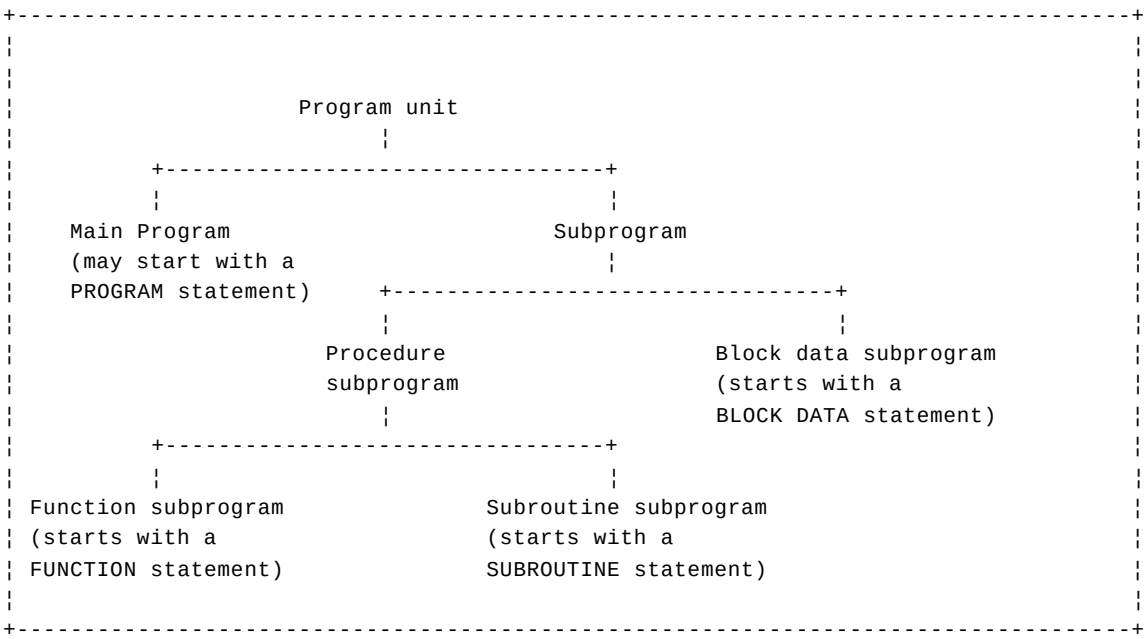


Figure 1. Program Unit Relationships

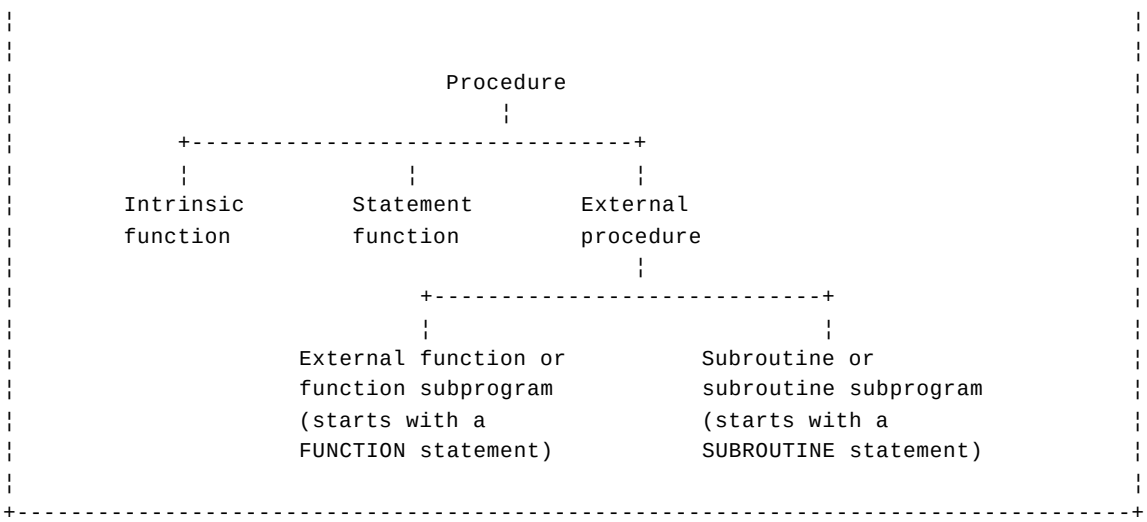


Figure 2. Procedure Relationships

10.2 PROGRAM Statement--Main Program

MVS	VM	OS/400	OS/2
X	X	X	X

name
is the name of a main program.

The PROGRAM statement specifies that a program unit is a main program. The PROGRAM statement is optional. A **main program** is the program unit that receives control from the system when the executable program is invoked at run time. A main program may contain any statement except BLOCK DATA, FUNCTION, SUBROUTINE, ENTRY, or RETURN.

Example of a PROGRAM Statement

```
        PROGRAM SCALE
```

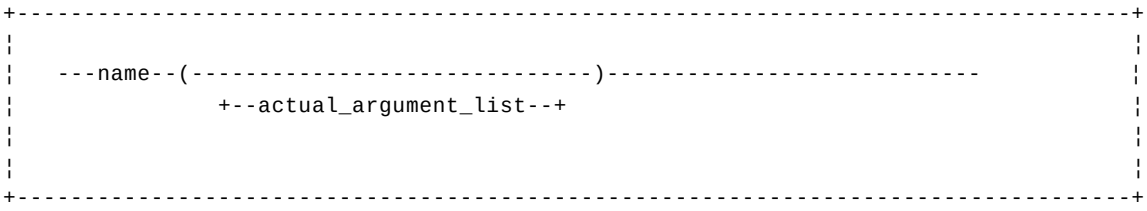
10.3 Functions

A **function** is a procedure that is invoked by its name or one of its entry names in a function reference and that returns a value to the point of reference. The three kinds of functions are intrinsic functions (see Appendix A, "Intrinsic Functions"), statement functions, and external functions (or function subprograms).

Subtopics

10.3.1 Function Reference

A **function reference** is used as a primary in an expression to invoke a function. The form of a function reference is:



name
is the name of an external function, an entry in an external function, a statement function, or an intrinsic function.

actual_argument
is an actual argument, described on page 10.8.

Execution of a function reference results in the following:

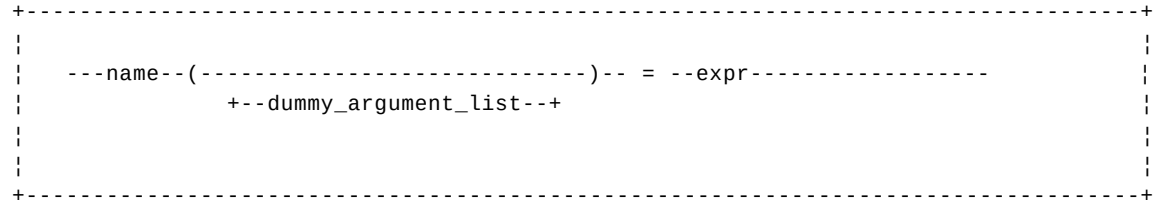
1. Actual arguments that are expressions are evaluated.
2. Actual arguments are associated with their corresponding dummy arguments.
3. The referenced function is executed.
4. The value of the function (called the **function value**) is available to the referencing expression.

Execution of a function reference must not alter the value of any other data item within the statement in which the function reference appears. However, execution of a function reference in the expression of a logical IF statement is permitted to affect data items in the statement that is executed when the value of the expression is true.

See "Examples of Statement Function Statements" in topic 10.3.2 and "Example of a FUNCTION Statement" in topic 10.3.3 for examples of function references.

10.3.2 Statement Function Statement

MVS	VM	OS/400	OS/2
X	X	X	X



name
is the name of this statement function.

dummy_argument
is a **statement function dummy argument**. See page 10.8 for a description of dummy arguments.

expr
is an expression.

A **statement function** is a single-statement function that is internal to the program unit in which it is defined. It is defined by a statement function statement and invoked by a function reference.

name determines the data type of the value returned from the statement function. If the data type of **name** does not match that of **expr**, the value of **expr** is converted to the type of **name**.

An external function reference in **expr** must not cause a **dummy_argument** of the statement function to become undefined or redefined.

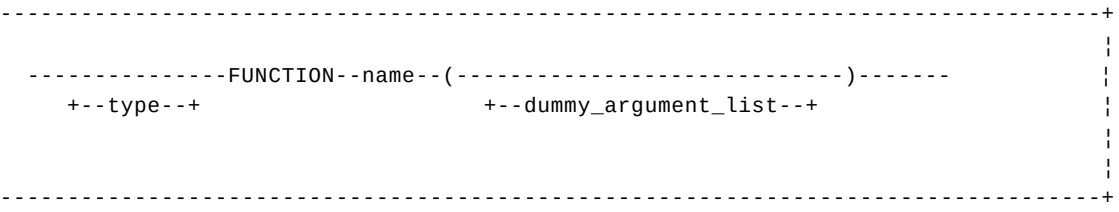
The name of a statement function of type character must not have a length specifier of an asterisk in parentheses.

Examples of Statement Function Statements

```
        parameter (pi = 3.14159)
        real area,circum,r,radius
  C Define statement functions AREA and CIRCUM.
        area(r) = pi * (r**2)
        circum(r) = 2 * pi * r
        .
        .
  C Reference the statement functions.
        print *,'The area is: ',area(radius)
        print *,'The circumference is: ',circum(radius)
```

10.3.3 FUNCTION Statement--Function Subprogram (External Function)

MVS	VM	OS/400	OS/2
X	X	X	X



type
explicitly specifies the data type of the value that the function subprogram returns. **type** may be INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, or CHARACTER[*__char_len__], where **char_len** is the length specification of the result of the character function. **char_len** may have any of the forms permitted in a CHARACTER type statement (see page 6.4), except that an integer constant expression must not include the name of a constant. The default is 1. The length associated with the function name in the function reference must be the same as **char_len**.

See "How Type Is Determined" in topic 3.2 for information on implicit typing.

name
is the name of this function subprogram. **name** may appear in a type statement, but in no other nonexecutable statement.

dummy_argument
is a dummy argument, described on page 10.8.

A FUNCTION statement specifies that a program unit is a function subprogram. A **function subprogram**, or **external function**, is a program unit that specifies a function. A function subprogram is invoked by a function reference and returns a value to the invoking program unit. For the purpose of returning the function value, the function name and any entry names are considered to be variable names, and you must assign a value to one of those names during every execution of the function.

The first statement of a function subprogram must be a FUNCTION statement. A function subprogram may contain any statement except PROGRAM, SUBROUTINE, and BLOCK DATA.

The variable whose name is the name of the function is associated with any variables whose names are also entry names. This is called **entry association**. The definition of any one of them becomes the definition of all the associated variables having that same type, and is the value of the function no matter at which entry point it was entered. Such variables are not required to be of the same type unless the type is character, but the variable whose name is used to reference the function must be in a defined state when a RETURN or END statement is executed in the subprogram. An associated variable of a different type must not become defined during the execution of the function reference.

Example of a FUNCTION Statement

Main Program

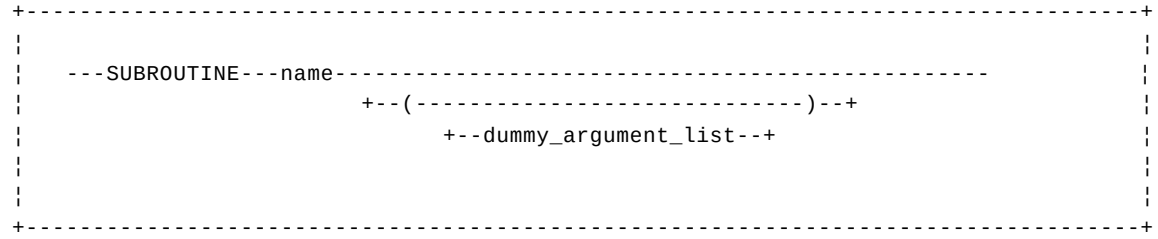
```
      program main
c Actual args are X2, X1, X0
      real root,x2,x1,x0
      .
      .
c     2*(x**2) + 4.5*x + 1
      x2 = 2.0
      x1 = 4.5
      x0 = 1.0
c Reference function sub.
      root = quad(x2,x1,x0)
      .
      .
```

Function Subprogram

```
c Dummy args are A, B, and C
      real function quad(a,b,c)
      real a,b,c
      quad = (-b + sqrt(b**2-4*a*c)) / (2*a)
      return
      end
```

10.4 SUBROUTINE Statement

MVS	VM	OS/400	OS/2
X	X	X	X



name
is the name of this subroutine subprogram.

dummy_argument
is a dummy argument, described on page 10.8.

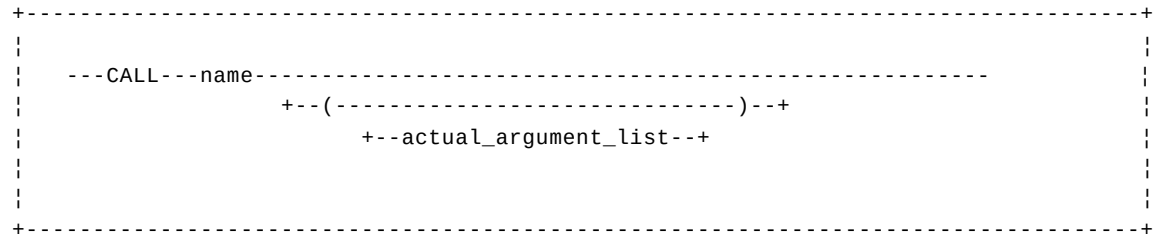
The SUBROUTINE statement specifies that a program unit is a subroutine subprogram. A **subroutine subprogram**, or **subroutine**, is a program unit that is invoked by its name or one of its entry names in a CALL statement. A subroutine subprogram may contain any statement except PROGRAM, FUNCTION, and BLOCK DATA.

Example of a SUBROUTINE Statement

```
        subroutine fit(j,e,b)
```

10.5 CALL Statement

MVS	VM	OS/400	OS/2
X	X	X	X



name
is the name of a subroutine or an entry in a subroutine.

actual_argument
is an actual argument, described on page 10.8.

The CALL statement:

1. Evaluates actual arguments that are expressions
2. Associates actual arguments with their corresponding dummy arguments
3. Invokes the specified subroutine.

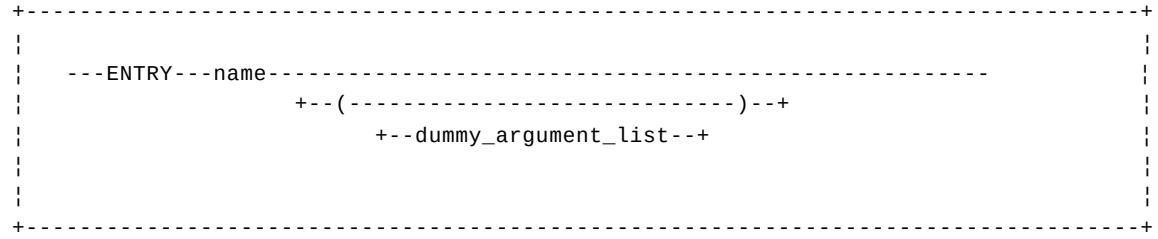
When the subroutine has processed, control returns from the subroutine.

Example of a CALL Statement

```
        call fit(k,e,q)
```

10.6 ENTRY Statement

MVS	VM	OS/400	OS/2
X	X	X	X



name
is the name of an entry in a function subprogram or subroutine subprogram and is called an **entry name**.

dummy_argument
is a dummy argument, described on page 10.8.

The ENTRY statement establishes an alternate entry point. A function subprogram or subroutine subprogram has a primary entry point that is established via the SUBROUTINE or FUNCTION statement.

In a function subprogram, **name** identifies an external function and may be referenced (invoked) as an external function. In a subroutine subprogram, **name** identifies a subroutine and may be referenced as a subroutine. When the reference is made, execution begins with the first executable statement following the ENTRY statement.

The name of an entry in a function subprogram must appear as a variable name in the function subprogram and must be defined upon exit from the subprogram.

In a function subprogram, **name** may appear in a type statement. In a function subprogram, **name** may be used as a variable name if the variable does not precede the ENTRY statement.

If an entry name in a function subprogram is of type character, all entry names in the subprogram and the name of the subprogram must be of type character. If the length specifier of an entry named in the function subprogram or the name of the subprogram itself is an asterisk in parentheses (indicating inherited length), all entry names and the subprogram name must have a length specifier of an asterisk in parentheses; otherwise, all such names must have a length specification of the same integer value.

A name in **dummy_argument_list** must not also appear:

In an executable statement preceding the ENTRY statement unless i also appears in a FUNCTION, SUBROUTINE, or ENTRY statement that precedes the executable statement

In the expression of a statement function statement unless the name i also a dummy argument of the statement function, appears in a FUNCTION

or SUBROUTINE statement, or appears in an ENTRY statement that precedes the statement function statement.

The number of dummy arguments and their data types in the **dummy_argument_list** of this ENTRY statement, of other ENTRY statements, and of the primary entry point, may differ.

Example of an ENTRY Statement

```
real function vol(rds,hgt)
parameter (pi = 3.14159)
real rds,hgt
a(rds) = pi * rds**2
vol= a(rds) * hgt
return
entry area(rds)
area = a(rds)
return
end
```

10.7 RETURN Statement

MVS	VM	OS/400	OS/2
X	X	X	X

integer_expr
is an integer expression.

The RETURN statement:

In a function subprogram, ends the execution of the subprogram an transfers control back to the referencing statement. The value of the function is available to the referencing program unit.

In a subroutine subprogram, ends the execution of the subprogram an transfers control to the first executable statement after the CALL statement or to an alternate return point, if one is specified.

integer_expr may be specified in a subroutine subprogram only, not a function subprogram, and it specifies an alternate return point. Letting **m** be the value of **integer_expr**, if 1 = **m** = the number of asterisks in the SUBROUTINE or ENTRY statement, the **m**th asterisk in the dummy argument list is selected. Control then returns to the invoking program unit at the statement whose statement label is specified in the **m**th alternate return specifier in the CALL statement. For example, if the value of **m** is 5, the fifth asterisk in the dummy argument list is selected, and control returns to the statement whose statement label is specified in the fifth alternate return specifier in the CALL statement.

If **integer_expr** is omitted or if its value (**m**) is not in the range one through the number of asterisks in the SUBROUTINE or ENTRY statement, a normal return is executed. Control returns to the invoking program unit at the statement following the CALL statement.

10.8 Arguments

An **actual argument** appears in the argument list of a procedure reference. An actual argument may be one of the following:

An expression, excluding a character expression involvin concatenation of an operand whose length specifier is an asterisk in parentheses (indicating inherited length)
An array nam
An intrinsic function name except for those listed under "INTRINSIC Statement" in topic 6.8
An external procedure nam
A dummy procedure nam
If the actual argument is in a CALL statement, an **alternate return specifier**, having the form ***stmt_label**, where **stmt_label** is the statement label of an executable statement.

A **dummy argument** appears in the argument list of a procedure. A dummy argument is specified in a statement function statement, FUNCTION statement, SUBROUTINE statement, or ENTRY statement. Statement functions, function subprograms, and subroutine subprograms use dummy arguments to indicate the types of actual arguments and whether each argument is a single value, array of values, procedure, or statement label. A dummy argument is classified as one of the following:

A variable nam
An array name (except in statement functions
A procedure name (except in statement functions
An asterisk (in subroutines only, to indicate an alternate retur point).

A given name may appear only once in a dummy argument list.

A dummy argument name must not be used in an EQUIVALENCE, DATA, PARAMETER, SAVE, or INTRINSIC statement. A dummy argument name must not be the same as the procedure name appearing in a FUNCTION, SUBROUTINE, ENTRY, or statement function statement in the same program unit.

A character dummy argument of inherited length must not be used as an operand for concatenation, except in a character assignment statement.

See "Example of a FUNCTION Statement" in topic 10.3.3 for an example of arguments.

Subtopics

10.8.1 Association of Arguments

Actual arguments are associated with dummy arguments when a function or subroutine is referenced (invoked). The first actual argument becomes associated with the first dummy argument, the second actual argument with the second dummy argument, and so forth. Argument association within a program unit terminates at the execution of a RETURN or END statement in the program unit. There is no retention of argument association between one reference of a subprogram and the next reference of the subprogram.

Actual arguments must agree in number, order, and type with their corresponding dummy arguments, except for two cases: a subroutine name has no type and must be associated with a dummy procedure name, and an alternate return specifier has no type and must be associated with an asterisk.

Argument association may be carried through more than one level of procedure reference.

If a subprogram reference causes a dummy argument in the referenced subprogram to become associated with another dummy argument in the referenced subprogram, neither dummy argument may become defined during execution of that subprogram. For example, if a subroutine is headed by:

```
  subroutine XYZ (A,B)
```

and is referenced by:

```
  call XYZ (C,C)
```

then the dummy arguments A and B each become associated with the same actual argument C and therefore with each other. Neither A nor B may become defined during this execution of subroutine XYZ or by any procedures referenced by XYZ.

If a subprogram reference causes a dummy argument to become associated with a data item in a common block in the referenced subprogram or in a subprogram referenced by the referenced subprogram, neither the dummy argument nor the data item in the common block may become defined within the subprogram or within a subprogram referenced by the referenced subprogram.

10.8.2 Length of Character Arguments

If arguments are of type character, the lengths of the actual arguments must be greater than or equal to the lengths of the dummy arguments. If an actual argument is longer, only the leftmost characters are associated with the dummy argument.

If a dummy argument has a length specifier of an asterisk in parentheses, the length of the dummy argument is "inherited" from the actual argument. The length is inherited because it is specified outside the program unit containing the dummy argument. If the associated actual argument is an array name, the length inherited by the dummy argument is the length of an array element in the associated actual argument array.

10.8.3 Variables As Dummy Arguments

A dummy argument that is a variable name must be associated with an actual argument that is an expression.

A dummy argument that is a variable name may be defined within a subprogram if the associated actual argument is a variable name, array element name, or character substring name. A dummy argument that is a variable name must not be redefined within a subprogram if the associated actual argument is a constant, name of a constant, function reference, expression involving operators, or expression enclosed in parentheses.

10.8.4 Arrays As Dummy Arguments

A dummy argument that is an array name must be associated with an actual argument that is an array name, an array element name, or an array element substring name. The number and size of the dimensions may differ.

If the actual argument is a noncharacter array name, the size of the dummy argument must not exceed the size of the actual argument and each actual array element is associated with the dummy array element of the same subscript value.

If the actual argument is a noncharacter array element name with a subscript value ***as***, the size of the dummy argument array must not exceed the size of the actual argument array plus one minus ***as*** and the dummy argument array element with a subscript value of ***ds*** becomes associated with the actual argument array element that has a subscript value of ***as*** + ***ds*** - 1.

If the actual argument is a character array name, character array element name, or array element substring name and begins at character storage unit ***acu*** of an array, character storage unit ***dcu*** of an associated dummy argument array becomes associated with character storge unit ***acu*** + ***dcu*** - 1 of the actual argument array.

10.8.5 Procedures As Dummy Arguments

A dummy argument that is a procedure is called a **dummy procedure**. A dummy procedure may only be associated with an actual argument that is an intrinsic function, external function, subroutine, or another dummy procedure.

The following example illustrates the use of a dummy procedure:

```
subroutine roots
external neg
.
.
x = quad(a,b,c,neg)
.
.
return
end

function quad(a,b,c,funct)
.
.
val = funct(a,b,c)
.
.
return
end

function neg(a,b,c)
.
.
return
end
```

10.8.6 Asterisks As Dummy Arguments

A dummy argument that is an asterisk may appear only in the dummy argument list of a SUBROUTINE statement or an ENTRY statement in a subroutine subprogram. The corresponding actual argument must be an alternate return specifier.

10.9 BLOCK DATA Statement--Block Data Subprogram

MVS	VM	OS/400	OS/2
X	X	X	X

name
is the name of this block data subprogram.

The BLOCK DATA statement specifies that a program unit is a block data subprogram. A **block data subprogram** is a program unit that provides initial values for variables and array elements in named common blocks. The only other statements that may appear in a block data subprogram are DIMENSION, EQUIVALENCE, COMMON, type, IMPLICIT, PARAMETER, SAVE, DATA, and END. Comment lines are permitted.

More than one block data subprogram is permitted in an executable program, but only one may be unnamed. More than one named common block may be initialized in a block data subprogram. Restrictions on common blocks in block data subprograms are:

All items in a named common block must appear in the COMMON statemen even though they are not all initialized.

The same named common block must not be referenced in two differen block data subprograms.

Only items in named common blocks may be initialized in block dat subprograms.

11.0 Chapter 11. Input/Output Statements

This chapter describes:

- Record
- File
- Unit
- The input/output statements: READ, WRITE, PRINT, OPEN, CLOSE, INQUIRE BACKSPACE, ENDFILE, and REWIND.

Subtopics

11.1 Records

A **record** is a sequence of characters or a sequence of values. The three kinds of records are formatted, unformatted, and endfile.

Subtopics

11.1.1 Formatted Records

A **formatted record** is a sequence of characters. When a formatted record is read, data values represented by characters are converted to internal form. When a formatted record is written, the data to be written is converted from internal form to characters.

If a formatted record is printed, (2) the first character of the record determines vertical spacing and is not printed. The remaining characters of the record, if any, are printed beginning at the left margin. Vertical spacing is as follows:

First Character of Record	**Vertical Spacing before Printing**
Blank	One line
0	Two lines
1	To first line of next page
+	No advance

(2) Printing may be performed on a printer or on some other device.

11.1.2 Unformatted Records

An **unformatted record** is a sequence of values in a system-dependent form and may contain both character and noncharacter data or may contain no data. The values are in their internal form and are not converted in any way when read or written.

11.1.3 Endfile Records

An **endfile record** is the last record of a file. It is written by an ENDFILE statement and has no length.

11.2 Files

A **file** is a sequence of records. The two kinds of files are external and internal. Access to an external file may be sequential or direct.

Subtopics
11.2.1 External Files
11.2.2 External File Access--Sequential or Direct
11.2.3 Internal Files

11.2.1 External Files

An **external file** is a file stored on an input/output device such as a disk, tape, or terminal.

An external file is said to **exist** for a program if it is available to the program for reading or was created within the program. Creating an external file causes it to exist when it did not previously. Deleting an external file ends its existence. An external file may exist but contain no records, if none were written yet. All input/output statements may refer to external files that exist. All input/output statements except READ may refer to external files that do not exist.

An external file may have a name. The name is system-dependent.

The **position** of an external file is usually established by the preceding input/output operation. An external file may be positioned to:

An **initial point**, which is the position just before the first record.

A **terminal point**, which is the position just after the last record.

A **current record**, when the file is positioned within a record. Otherwise, there is no current record.

A **preceding record**, which is the record just before the current file position. A preceding record does not exist when the file is positioned at its initial point or at the first record of the file.

A **next record**, which is the record just after the current file position. The next record does not exist when the file is positioned at the terminal point or in the last record of the file.

An indeterminate position after an error

11.2.2 External File Access--Sequential or Direct

The two methods of accessing the records of an external file are sequential and direct. The method is determined when the file is connected to a unit.

A file connected for **sequential access** contains records in the order they were written. The records must be either all formatted or all unformatted, except that the last record of the file must be an endfile record. The records must not be read or written by direct access input/output statements during the time the file is connected for sequential access.

The records of a file connected for **direct access** may be read or written in any order. The records must be either all formatted or all unformatted, except that the last record of the file may be an endfile record if the file may also be connected for sequential access. In this case, however, the endfile record is not considered to be part of the file while the file is connected for direct access. The records must not be read or written by sequential access input/output statements during the time the file is connected for direct access, or read or written using list-directed formatting.

Each record in a file connected for direct access has a **record number**, which identifies its order in the file. The record number is an integer value that must be specified when the record is read or written. Records are numbered sequentially. The first record is number 1. Records need not be read or written in the order of their record numbers. For example, records 9, 5, and 11 can be written in that order without writing the intermediate records.

All records in a file connected for direct access must have the same length, which is specified when the file is connected.

Records in a file connected for direct access cannot be deleted but can be rewritten with a new value. A record cannot be read unless it was first written.

11.2.3 Internal Files

An **internal file** is a character variable, character array, character array element, or character substring.

If an internal file is a character variable, character array element, or character substring, the file consists of one record with a length equal to that of the variable, array element, or substring. If an internal file is a character array, each element of the array is a record of the file, with each record having the same length.

Reading and writing records is accomplished only by sequential-access formatted input/output statements that do not specify list-directed formatting. READ, WRITE, and PRINT are the only input/output statements that may specify an internal file.

If a WRITE statement writes less than an entire record, blanks fill the remainder of the record.

An internal file always exists.

A variable, array element, or character substring that is a record of an internal file may become defined or undefined by means other than an output statement. For example, it may become defined by a character assignment statement.

11.3 Units

A **unit** is a means of referring to an external file. Programs refer to external files by the unit numbers specified in unit specifiers in input/output statements. See page 11.4 for the form of a unit specifier.

Subtopics

11.3.1 Connection of a Unit

The association of a unit with an external file is called a **connection**. Connection must occur before the records of the file may be read or written. Connection may occur by **preconnection**, which is prior to program execution, or by an OPEN statement. See the publications for your FORTRAN product for more information about preconnection.

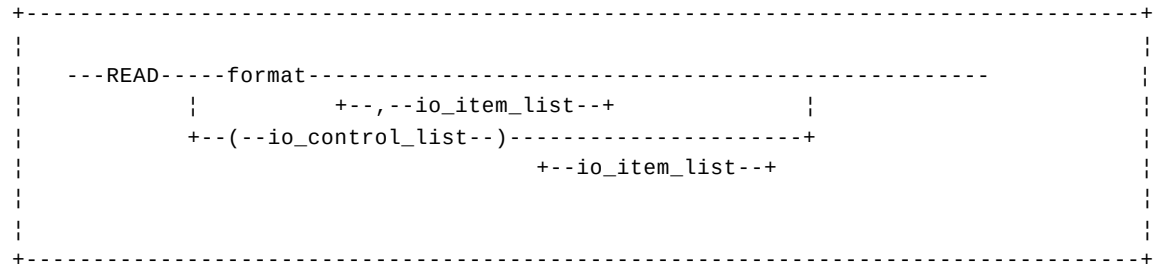
A file may be connected and not exist. An example is a preconnected new file.

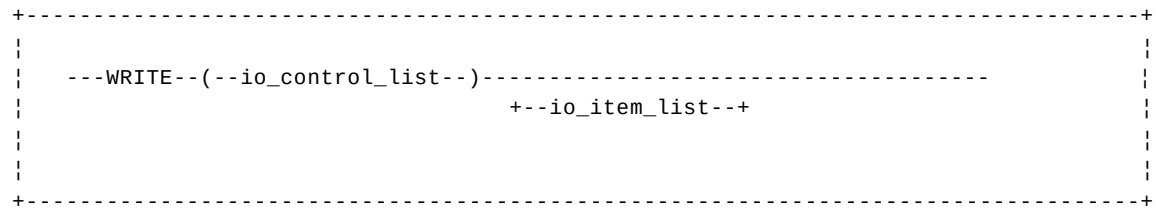
All input/output statements except OPEN, CLOSE, and INQUIRE must specify units that are connected to an external file.

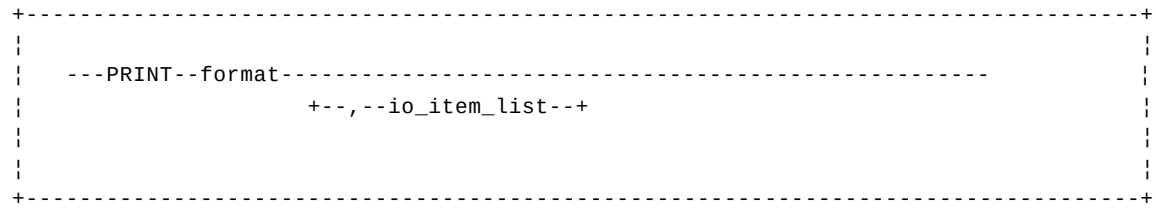
The CLOSE statement disconnects a file from a unit. The file may be connected again within the same executable program to the same unit or to a different unit, and the unit may be connected again within the same executable program to the same file or to a different file.

11.4 READ, WRITE, and PRINT Statements

MVS	VM	OS/400	OS/2
X	X	X	X







format
is a format identifier, described below under **FMT=format**.

io_item
is an input/output list item. An **input/output list** specifies the data to be transferred. An input/output list item may be:

A variable name.
An array element name.
A character substring name.
An array name. The array is treated as if all of its elements were specified in the order they are arranged in storage.
In an output list only, any other expression except a character expression involving concatenation of an operand whose length specifier is an asterisk in parentheses (indicating inherited length) unless the operand is the name of a constant.
An implied-DO list, described on page 11.4.4.

io_control_list
is a list that must contain one unit specifier and may also contain one of each of the other permitted specifiers. The permitted specifiers are:

[**UNIT=]u**
is a **unit specifier**, which specifies the unit to be used in the input/output operation. **u** is an external unit identifier or internal file identifier.

An **external unit identifier** refers to an external file. It is one of the following:

An INTEGER*4 expression whose value is in the range zero through 99, inclusive.
An asterisk, identifying an installation-defined unit that is preconnected for formatted sequential access. **Note:** Although other input/output statements also allow a unit specifier, only the READ, WRITE, and PRINT statements allow its value to be an asterisk.

An **internal file identifier** refers to an internal file. It is the name of a character variable, character array, character array element, or character substring.

If the optional characters UNIT= are omitted, **u** must be the first item in **io_control_list**.

[**FMT=]format**
is a **format specifier**, which specifies the format to be used in the input/output operation. **format** is a **format identifier**, which may be:

The statement label of a FORMAT statement. (The FORMAT statement is described on page 12.1.2.)
The name of an INTEGER*4 variable that was assigned the statement label of a FORMAT statement.
The name of a character array. (See "Character Format Specification" in topic 12.1.3 for more information.)
Any character expression except one involving concatenation of an operand whose length specifier is an asterisk in parentheses (indicating inherited length) unless the operand is the name of a constant. (See "Character Format Specification" in topic 12.1.3 for more information.)
An asterisk, specifying list-directed formatting. (List-directed formatting is described on page 12.4.)

If the optional characters FMT= are omitted, **format** must be the second item in **io_control_list** and the first item must be the unit specifier with UNIT= omitted.

REC=integer_expr
is a **record specifier**, which specifies the number of the record to be read or written in a file connected for direct access. **integer_expr** is an integer expression whose value is positive. A record specifier is not permitted if formatting is list-directed, if the unit specifier specifies an internal file, or if an end-of-file specifier is specified.

IOSTAT=ios
is an **input/output status specifier**, which specifies the status of the input/output operation. **ios** is the name of a variable or array element of type INTEGER*4. When the input/output statement containing this specifier finishes execution, **ios** is defined with:

(For a READ statement only) a negative value if an end-of-file specifier is specified, an end-of-file condition was encountered, and no error occurred during execution of the READ statement.

A zero value if the input/output operation completed without any errors.

A positive value if an error occurred during the input/output operation and an error specifier is specified. The meaning of a positive value is system-dependent.

ERR=stmt_label
is an **error specifier**, which specifies a statement label at which execution is to continue when an error occurs during the execution of the input/output statement.

END=stmt_label
is an **end-of-file specifier**, which specifies a statement label at which execution is to continue when an endfile record is encountered while reading from a file and no error occurred. This specifier may only be specified in a READ statement that refers to a unit connected for sequential access. If an end-of-file specifier is specified, a record specifier is not permitted.

A READ statement without **io_control_list** specified specifies the same unit as a READ statement with **io_control_list** specified in which the external unit identifier is an asterisk.

Subtopics

11.4.1 Categories of READ, WRITE, and PRINT Statements

A READ or WRITE statement may be a formatted input/output statement or an unformatted input/output statement. The PRINT statement is a formatted input/output statement.

A **formatted input/output statement** contains a format identifier and transfers data with editing (conversion) occurring between the internal form of the data and the character representation of that data in records. The two methods of formatting are:

Format-directed formatting, where editing is controlled by edi descriptors in a format specification. Format specifications are described on page 12.1.1.

List-directed formatting, where editing is controlled by the types an lengths of the data being read or written. List-directed formatting is described on page 12.4.

If a formatted READ, WRITE, or PRINT statement has an asterisk as a format identifier, the statement is a **list-directed input/output statement**, and a record specifier must not be present.

An **unformatted input/output statement** does not contain a format identifier and transfers data without performing editing.

A READ or WRITE statement is a **direct access input/output statement** if it contains a record specifier, or a **sequential access input/output statement** if it does not contain a record specifier.

11.4.2 Execution of READ, WRITE, and PRINT Statements

The READ statement reads data from an external file to internal storage or from an internal file to internal storage. Values are transferred from the file to the data items specified by the input list (**io_item_list**), if one is specified.

The WRITE and PRINT statements write data from internal storage to an external file or from internal storage to an internal file. Values are transferred to the file from the data items specified by the output list (**io_item_list**) and format specification, if they are specified. Execution of a WRITE or PRINT statement for a file that does not exist creates the file, unless an error occurs.

11.4.3 File Position before and after Data Transfer

The positioning of a file prior to data transfer depends on the method of access:

Sequential access for an external file: On input, the file i positioned at the beginning of the next record. This record becomes the current record. On output, a new record is created and becomes the last record of the file.

Sequential access for an internal file: The file is positioned at the beginning of the first record of the file. This record becomes the current record.

Direct access: The file is positioned at the beginning of the recor specified by the record specifier. This record becomes the current record.

After data transfer, the file is positioned:

Beyond the endfile record if an end-of-file condition exists as result of reading an endfile record.

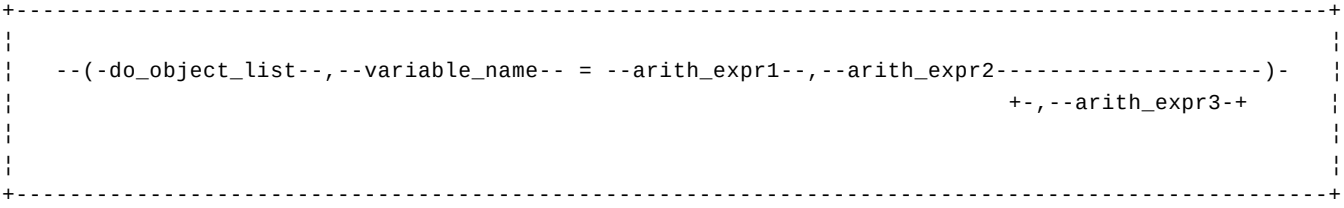
Beyond the last record read or written if no error or end-of-fil condition exists. That last record becomes the preceding record. A record written on a file connected for sequential access becomes the last record of the file.

If a file is positioned beyond the endfile record, execution of a READ, WRITE, PRINT, or ENDFILE statement is not permitted. However, a BACKSPACE or REWIND statement may be used to reposition the file.

If an error occurs, the position of an external file is indeterminate.

11.4.4 Implied-DO List in a READ, WRITE, or PRINT Statement

An implied-DO list may be used in a READ, WRITE, or PRINT statement to specify the data to be transferred. Its form is:



do_object
is an input/output list item (see page 11.4).

variable_name
arith_expr1
arith_expr2
arith_expr3
are as specified for the DO statement (see page 9.7).

The range of an implied-DO list is the list **do_object_list**. The iteration count and the values of the DO variable are established from **arith_expr1**, **arith_expr2**, and **arith_expr3**, the same as for a DO statement. (See "Execution of a DO Statement" in topic 9.7.3.) When the implied-DO list is executed, the items in the **do_object_list** are specified once for each iteration of the implied-DO list, with the appropriate substitution of values for any occurrence of the DO variable.

In a READ statement, the DO variable or an associated data item must not appear as an input list item in the **do_object_list**, but may be read in the same READ statement outside of the implied-DO list. For example:

```
        read(3,150) isize,(jinx(i),i=1,isize)
   150  format(10i7)
```

In the example, the value of ISIZE is read with the same READ statement but outside of the implied-DO list of which it is a part. One element of the array JINX is defined with each iteration of the implied-DO list.

11.4.5 Examples of READ, WRITE, and PRINT Statements

Example of Formatted READ and WRITE Statements

```
        integer length,width,depth
        character*8 chr_time
        .
        .
        read(10,200) length,width,depth
  200   format(i5,i10,i10)
        write(*,'(a,a)') 'The time is:',chr_time(1:8)
```

Example of Unformatted READ and WRITE Statements

```
        integer data_unit,size,a(1000),buffer(2000)
        .
        .
        read(unit=data_unit) size,(a(j),j=1,size)
        write(20) buffer
```

11.5 OPEN Statement

MVS	VM	OS/400	OS/2
X	X	X	X

open_list
is a list that must contain one unit specifier (UNIT=**u**) and may also contain one of each of the other permitted specifiers. The permitted specifiers are:

[**UNIT=]u**
is a unit specifier in which **u** must be an external unit identifier whose value is not an asterisk. External unit identifiers are described on page 11.4. If the optional characters UNIT= are omitted, **u** must be the first item in **open_list**.

IOSTAT=ios
is an input/output status specifier, described on page 11.4.

ERR=stmt_label
is an error specifier, described on page 11.4.

FILE=char_expr
is a **file specifier**, which specifies the name of the file to be connected to the specified unit. **char_expr** is a character expression whose value, when any trailing blanks are removed, is the system-dependent name of the file. If the file specifier is omitted, the unit becomes connected to a system-determined file.

STATUS=char_expr
specifies the status of the file when it is opened. **char_expr** is a character expression whose value, when any trailing blanks are removed, is one of the following:

OLD, to connect an existing file to a unit. If OLD is specified, a file specifier must be specified.
NEW, to create a new file and connect it to a unit. If NEW is specified, a file specifier must be specified.
SCRATCH, to create and connect a new file that will be deleted when it is disconnected. SCRATCH must not be specified with a named file (that is, FILE=**char_expr** must be omitted).
UNKNOWN, to connect an existing file, or to create and connect a new file. If the file exists it is connected as OLD. If the file does not exist it is connected as NEW.

UNKNOWN is the default.

ACCESS=char_expr
specifies the access method for the connection of the file.

char_expr is a character expression whose value, when any trailing blanks are removed, is either SEQUENTIAL or DIRECT. SEQUENTIAL is the default.

FORM=char_expr
specifies whether the file is connected for formatted or unformatted input/output. **char_expr** is a character expression whose value, when any trailing blanks are removed, is either FORMATTED or UNFORMATTED. If the file is being connected for sequential access, FORMATTED is the default. If the file is being connected for direct access, UNFORMATTED is the default.

RECL=integer_expr
specifies the length of each record in a file being connected for direct access. **integer_expr** is an INTEGER*4 expression whose value must be positive. This specifier must be omitted when a file is being connected for sequential access.

BLANK=char_expr
controls the default interpretation of blanks when using a format specification. **char_expr** is a character expression whose value, when any trailing blanks are removed, is either NULL or ZERO. See "BN (Blank Null) and BZ (Blank Zero) Editing" in topic 12.3.5 for descriptions of NULL and ZERO.

The OPEN statement may be used to connect an existing external file to a unit, create an external file that is preconnected, create an external file and connect it to a unit, or change certain specifiers of a connection between an external file and a unit.

If a unit is connected to a file that exists, execution of an OPEN statement for that unit is permitted. If the file specifier is not included in the OPEN statement, the file to be connected to the unit is the same as the file to which the unit is connected.

If the file to be connected to the unit does not exist, but is the same as the file to which the unit is preconnected, the properties specified by the OPEN statement become a part of the connection.

If the file to be connected to the unit is not the same as the file to which the unit is connected, the effect is as if a CLOSE statement without a STATUS=**char_expr** specifier had been executed for the unit immediately prior to the execution of the OPEN statement.

If the file to be connected to the unit is the same as the file to which the unit is connected, only the BLANK=**char_expr** specifier may have a value different from the one currently in effect. Execution of the OPEN statement causes the new value of the BLANK=**char_expr** specifier to be in effect. The position of the file is unaffected.

If a file is connected to a unit, execution of an OPEN statement on that file and a different unit is not permitted.

Example of an OPEN Statement

```
character*20 fname
fname = 'input.dat'
open(unit=8,file=fname,status='new',form='formatted')
```

In the example, the value of character variable FNAME is system-dependent.

11.6 CLOSE Statement

MVS	VM	OS/400	OS/2
X	X	X	X

close_list
is a list that must contain one unit specifier (UNIT=**u**) and may also contain one of each of the other permitted specifiers. The permitted specifiers are:

[**UNIT=]u**
is a unit specifier in which **u** must be an external unit identifier whose value is not an asterisk. External unit identifiers are described on page 11.4. If the optional characters UNIT= are omitted, **u** must be the first item in **close_list**.

IOSTAT=ios
is an input/output status specifier, described on page 11.4.

ERR=stmt_label
is an error specifier, described on page 11.4.

STATUS=char_expr
specifies the disposition of the file after it is closed. **char_expr** is a character expression whose value, when any trailing blanks are removed, is either KEEP or DELETE.

If KEEP is specified for a file that exists, the file will continue to exist after the execution of the CLOSE statement. If KEEP is specified for a file that does not exist, the file will not exist after the execution of the CLOSE statement. KEEP must not be specified for a file whose status prior to execution of the CLOSE statement is SCRATCH.

If DELETE is specified, the file will not exist after execution of the CLOSE statement.

The default is DELETE if the file status prior to execution of the CLOSE statement is SCRATCH; otherwise it is KEEP.

The CLOSE statement disconnects an external file from a unit.

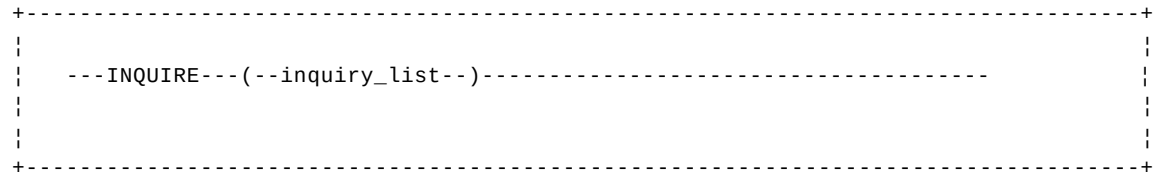
At termination of execution of an executable program for reasons other than an error condition, all units that are connected are closed. Each unit is closed with status KEEP unless the file status prior to termination of execution was SCRATCH, in which case the unit is closed with status DELETE. Note that the effect is as though a CLOSE statement without a STATUS=**char_expr** specifier were executed on each connected unit.

Examples of CLOSE Statements

```
close(15)
close(unit=16,status='delete')
```

11.7 INQUIRE Statement

MVS	VM	OS/400	OS/2
X	X	X	X



inquiry_list
is a list of **inquiry specifiers**. In an INQUIRE-by-file statement, **inquiry_list** must contain one file specifier (FILE=**char_expr**), must not contain a unit specifier (UNIT=**u**), and may contain at most one of each of the other inquiry specifiers. In an INQUIRE-by-unit statement, **inquiry_list** must contain one unit specifier, must not contain a file specifier, and may contain at most one of each of the other inquiry specifiers. The inquiry specifiers are:

FILE=char_expr
is a file specifier, and specifies the name of the file about which an INQUIRE-by-file statement is inquiring. **char_expr** is a character expression whose value, when any trailing blanks are removed, is the system-dependent name of the file. The named file does not have to exist nor does it have to be associated with a unit.

[**UNIT=]u**
is a unit specifier, and specifies the unit about which an INQUIRE-by-unit statement is inquiring. **u** must be an external unit identifier whose value is not an asterisk. External unit identifiers are described on page 11.4. If the optional characters UNIT= are omitted, **u** must be the first item in **inquiry_list**.

IOSTAT=ios
is an input/output status specifier, described on page 11.4.

ERR=stmt_label
is an error specifier, described on page 11.4. The INQUIRE statement does not cause any error conditions.

EXIST=ex
indicates whether a file or unit exists. **ex** is a logical variable or logical array element that is assigned the value true or false. For an INQUIRE-by-file statement, the value true is assigned if the file specified by FILE=**char_expr** exists, or the value false is assigned if the file does not exist. For an INQUIRE-by-unit statement, the value true is assigned if the unit specified by UNIT=**u** exists, or the value false is assigned if the unit does not exist.

OPENED=od
indicates whether a file or unit is connected. **od** is a logical

variable or logical array element that is assigned the value true or false. For an INQUIRE-by-file statement, the value true is assigned if the file specified by FILE=**char_expr** is connected to a unit, or the value false is assigned if the file is not connected to a unit. For an INQUIRE-by-unit statement, the value true is assigned if the unit specified by UNIT=**u** is connected to a file, or the value false is assigned if the unit is not connected to a file.

NUMBER=num
indicates the external unit identifier currently associated with the file. **num** is an INTEGER*4 variable or array element that is assigned the value of the external unit identifier of the unit that is currently connected to the file. If there is no unit connected to the file, **num** becomes undefined.

NAMED=nmd
indicates whether the file has a name. **nmd** is a logical variable or logical array element that is assigned the value true if the file has a name, or the value false if the file does not have a name.

NAME=fn
indicates the name of the file. **fn** is a character variable or character array element that is assigned the value of the name of the file if the file has a name, or becomes undefined if the file does not have a name.

ACCESS=char_expr
indicates whether the file is connected for sequential access or direct access. **char_expr** is a character variable or character array element that is assigned the value SEQUENTIAL if the file is connected for sequential access, or the value DIRECT if the file is connected for direct access. If there is no connection, **char_expr** becomes undefined.

SEQUENTIAL=seq
indicates whether the file can be accessed sequentially. **seq** is a character variable or character array element that is assigned the value YES if the file can be accessed sequentially, the value NO if the file cannot be accessed sequentially, or the value UNKNOWN if it cannot be determined.

DIRECT=dir
indicates whether the file can be accessed directly. **dir** is a character variable or character array element that is assigned the value YES if the file can be accessed directly, the value NO if the file cannot be accessed directly, or the value UNKNOWN if it cannot be determined.

FORM=char_expr
indicates whether the file is connected for formatted or unformatted input/output. **char_expr** is a character variable or character array element that is assigned the value FORMATTED if the file is connected for formatted input/output, or the value UNFORMATTED if the file is connected for unformatted input/output. If there is no connection, **char_expr** becomes undefined.

FORMATTED=fmt
indicates whether the file can be connected for formatted

input/output. **fmt** is a character variable or character array element that is assigned the value YES if the file can be connected for formatted input/output, the value NO if the file cannot be connected for formatted input/output, or the value UNKNOWN if it cannot be determined.

UNFORMATTED=unf
indicates whether the file can be connected for unformatted input/output. **fmt** is a character variable or character array element that is assigned the value YES if the file can be connected for unformatted input/output, the value NO if the file cannot be connected for unformatted input/output, or the value UNKNOWN if it cannot be determined.

RECL=rcl
indicates the record length of a file connected for direct access. **rcl** is an INTEGER*4 variable or array element that is assigned the value of the record length. If there is no connection or if the connection is not for direct access, **rcl** becomes undefined.

NEXTREC=nr
indicates where the next record may be read or written on a file connected for direct access. **nr** is an INTEGER*4 variable or array element that is assigned the value **n** + 1, where **n** is the record number of the last record read or written on the file connected for direct access. If the file is connected but no records were read or written since the connection, **nr** is assigned the value 1. If the file is not connected for direct access or if the position of the file cannot be determined because of a previous error, **nr** becomes undefined.

BLANK=char_expr
indicates the default treatment of blanks for a file connected for formatted input/output. **char_expr** is a character variable or character array element that is assigned the value NULL if all blanks in numeric input fields are ignored (as in BN editing), or the value ZERO if all nonleading blanks are interpreted as zeros (as in BZ editing). If there is no connection, or if the connection is not for formatted input/output, **char_expr** becomes undefined.

The INQUIRE statement obtains information about:

The properties of an external file. When used for this purpose th file specifier (FILE=**char_expr**) must be specified and the statement is called an **INQUIRE-by-file** statement.
An external file's association with a particular unit. When used fo this purpose the unit specifier (UNIT=**u**) must be specified and the statement is called an **INQUIRE-by-unit** statement.

An INQUIRE statement may be executed before, while, and after a file is associated with a unit. Any values assigned as the result of an INQUIRE statement are values that are current at the time the statement is executed.

An INQUIRE-by-file statement defines the **inquiry_list** variables and array elements as follows:

Variables or array elements specified by NAMED **nmd**, NAME=**fn**,

SEQUENTIAL=**seq**, DIRECT=**dir**, FORMATTED=**fmt**, and UNFORMATTED=**unf** become defined only if the value of **char_expr** is the name of a file that exists; otherwise, the variables or array elements become undefined.

A variable or array element specified by NUMBER **num** becomes defined only if a variable or array element specified by OPENED=**od** becomes defined with the value true.

Variables or array elements specified by ACCESS **char_expr**, FORM=**char_expr**, RECL=**rcl**, NEXTREC=**nr**, and BLANK=**char_expr** become defined only if a variable or array element specified by OPENED=**od** becomes defined with the value true.

An INQUIRE-by-unit statement defines the **inquiry_list** variables and array elements specified by NUMBER=**num**, NAMED=**nmd**, NAME=**fn**, ACCESS=**char_expr**, SEQUENTIAL=**seq**, DIRECT=**dir**, FORM=**char_expr**, FORMATTED=**fmt**, UNFORMATTED=**unf**, RECL=**rcl**, NEXTREC=**nr**, and BLANK=**char_expr** only if the specified unit exists and if a file is connected to the unit; otherwise, the variables or array elements become undefined.

If an error occurs during execution of an INQUIRE statement, all of the **inquiry_list** variables and array elements become undefined, except the one specified by IOSTAT=**ios**.

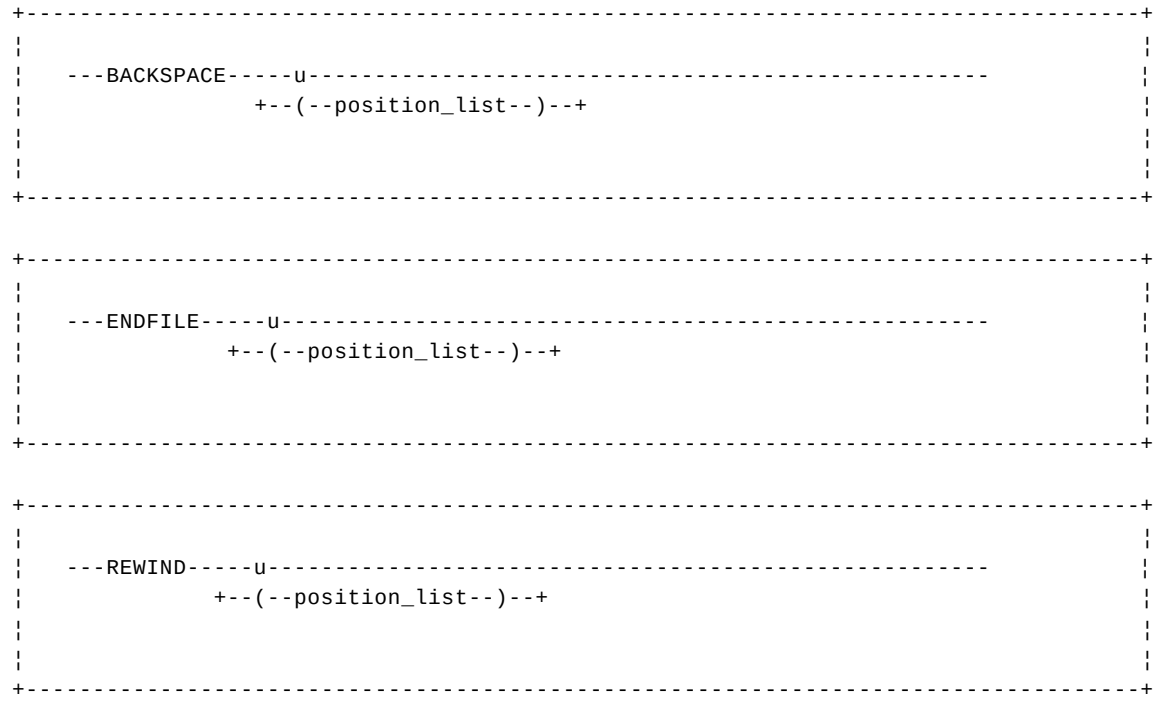
The **inquiry_list** variables or array elements specified by EXIST=**ex** and OPENED=**od** always become defined unless an error occurs.

Example of an INQUIRE Statement

```
inquire(file=file1,exist=f_ex,opened=f_od,number=f_num)
```

11.8 BACKSPACE, ENDFILE, and REWIND Statements

MVS	VM	OS/400	OS/2
X	X	X	X



u
is an external unit identifier, described on page 11.4. The value of **u** must not be an asterisk.

position_list
is a list that must contain one unit specifier (UNIT=**u**) and may also contain one of each of the other permitted specifiers. The permitted specifiers are:

[**UNIT=]u**
is a unit specifier in which **u** must be an external unit identifier whose value is not an asterisk. External unit identifiers are described on page 11.4. If the optional characters UNIT= are omitted, **u** must be the first item in **position_list**.

IOSTAT=ios
is an input/output status specifier, described on page 11.4.

ERR=stmt_label
is an error specifier, described on page 11.4.

External files connected for sequential access may be positioned using BACKSPACE, ENDFILE, and REWIND statements, with the following effects:

BACKSPACE positions a file connected to a specified unit before th preceding record. If there is no preceding record, the file position does not change. If the preceding record is the endfile record, the

file is positioned before the endfile record. Backspacing over records that were written using list-directed formatting is not permitted.

ENDFILE writes an endfile record as the next record of a file. Thi record becomes the last record in the file.

REWIND positions a file at its initial point

Examples of BACKSPACE, ENDFILE, and REWIND Statements

```
backspace 15
backspace (unit=15,err=99)
endfile 12
endfile (iostat=ioss,unit=11)
rewind 9
```

12.0 Chapter 12. Input/Output Formatting

Formatted READ, WRITE, and PRINT statements use formatting information to direct the editing (conversion) between internal data representations and character representations in formatted records (see "Formatted Records" in topic 11.1.1). This chapter describes the two methods of formatting:

Format-directed formattin
List-directed formatting

Subtopics

12.1 Format-Directed Formatting

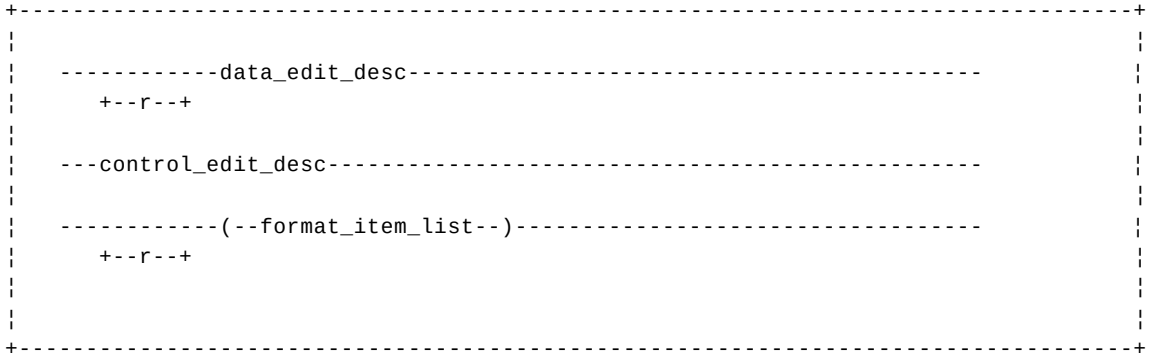
With format-directed formatting, editing is controlled by **edit descriptors** in a format specification. A **format specification** is specified in a FORMAT statement or as the value of a character array or character expression in a READ, WRITE, or PRINT statement.

Subtopics

12.1.1 Format Specification

A format specification (**format_spec**) has the form:

A **format_item** has any of the following forms:



r
is an unsigned, nonzero, integer constant called a **repeat specification**. The default is 1.

data_edit_desc
is a **data** (or **repeatable**) **edit descriptor**. The forms are:

Forms	**Use**	**See Page**
A Aw	Edits character values	12.3.3
Ew.d Ew.dEe Dw.d	Edits real and complex numbers with exponents	12.3.6
Fw.d	Edits real and complex numbers without exponents	12.3.7
Gw.d Gw.dEe	Edits real and complex numbers, with the output format adapting to the magnitude of the number	12.3.8
Iw Iw.m	Edits integer numbers	12.3.10

Lw	Edits logical values	12.3.11
Zw	Edits hexadecimal values	12.3.15

control_edit_desc
is a **control** (or **nonrepeatable**) **edit descriptor**. The forms are:

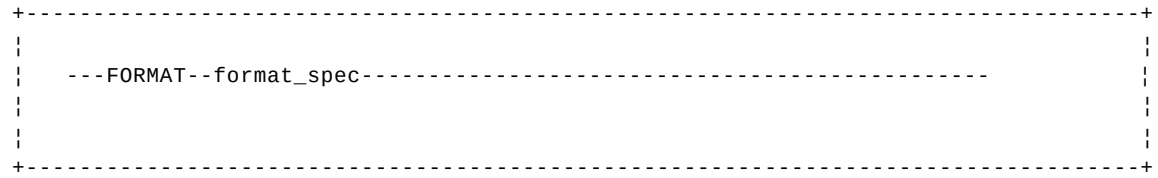
Forms	**Use**	**See Page**
/	Specifies the end of data transfer on the current record	12.3.1
:	Specifies the end of format control if there are no more items in the input/output list	12.3.2
'**h**'	Specifies a character string (**h**) for output	12.3.4
BN	Specifies that blanks in numeric input fields are to be ignored	12.3.5
BZ	Specifies that nonleading blanks in numeric input fields are to be interpreted as zeros	12.3.5
nHh	Specifies a character string for output	12.3.9
kP	Specifies a scale factor	12.3.12
S SS	Specifies that plus signs are not to be written	12.3.13
SP	Specifies that plus signs are to be written	12.3.13
T**c**	Specifies the absolute position in a record from which, or to which, the next character is transferred	12.3.14
TL**c**	Specifies the relative position (backward from the current position in a record) from which, or to which, the next character is transferred	12.3.14
TR**c**	Specifies the relative position (forward from the current position in a record) from which, or to which, the next character is transferred	12.3.14

nX	Specifies the relative position (forward from the current position in a record) from which, or to which, the next character is transferred	12.3.14

Commas separate edit descriptors. However, you may omit the comma between a P edit descriptor and an F, E, D, or G edit descriptor immediately following it; before or after a slash edit descriptor; and before or after a colon edit descriptor.

12.1.2 FORMAT Statement

MVS	VM	OS/400	OS/2
X	X	X	X



format_spec
is described on page 12.1.1.

The FORMAT statement specifies a format specification. When a format identifier (page 11.4) in a formatted READ, WRITE, or PRINT statement is a statement label or a statement label assigned to a variable name, the statement label identifies a FORMAT statement.

The FORMAT statement must have a statement label.

Examples of FORMAT Statements

```
990  format(i5, 2f10.2)
880  format(i5, f10.2, i5)
```

12.1.3 Character Format Specification

When a format identifier (page 11.4) in a formatted READ, WRITE, or PRINT statement is a character array name or character expression, the value of the array or expression is a character format specification. Such a format specification has the form **format_spec**, described on page 12.1.1.

If the format identifier is a character array element name, the format specification must be completely contained within the array element. If the format identifier is a character array name, the format specification may continue beyond the first element into following consecutive elements.

Blanks may precede the format specification. Character data may follow the right parenthesis that ends the format specification, with no effect on the format specification.

Example of a Character Format Specification

```
character*18 charvar
.
.
charvar = '(f10.2, i5, f10.2)'
write(*,charvar) solid, liquid, gas
```

12.2 Interaction between an Input/Output List and a Format Specification

The beginning of format-directed formatting initiates **format control**. Each action of format control depends on the next edit descriptor contained in the format specification and the next item in the input/output list, if one exists.

If an input/output list specifies at least one item, at least one data (repeatable) edit descriptor must exist in the format specification. Note that an empty format specification (parentheses only) may be used only if there are no items in the input/output list. In this case one input record is skipped or one output record containing no characters is written.

A format specification is interpreted from left to right except when a repeat specification (**r**) is present. A format item preceded by a repeat specification is processed as a list of **r** format specifications or edit descriptors identical to the format specification or edit descriptor without the repeat specification.

To each data (repeatable) edit descriptor interpreted in a format specification, there corresponds one item specified by the input/output list, except that a list item of type complex requires the interpretation of two F, E, D, or G edit descriptors. To each control (nonrepeatable) edit descriptor there is no corresponding item specified by the input/output list, and format control communicates information directly with the record.

Format control operates as follows:

1. If a data (repeatable) edit descriptor is encountered, format control processes an input/output list item if there is one, or terminates if the list is empty. If the list item processed is type complex, two F, E, D, or G edit descriptors are processed.

2. If a colon edit descriptor is encountered, format control processes an input/output list item if there is one, or terminates if the list is empty.

3. If a control (nonrepeatable) edit descriptor other than a colon is encountered, format control processes an input/output list item.

4. If the end of the format specification is reached, format control terminates if the input/output list is empty, or reverts to the beginning of the format specification terminated by the last preceding right parenthesis. Concerning reversion:

   The reused portion of the format specification must contain at least one data (repeatable) edit descriptor.

   If reversion is to a parenthesis that is preceded by a repeat specification, the repeat specification is reused.

   Reversion, of itself, has no effect on the scale factor; on the S, SP, or SS edit descriptors; or on the BN or BZ edit descriptors.

   If format control reverts, the file is positioned in a manner identical to the way it is positioned when a slash edit descriptor is processed.

During a read operation, any unprocessed characters of the record are skipped whenever the next record is read.

12.3 Editing

Editing is performed on fields. A **field** is the part of a record that is read on input or written on output when format control processes one I, F, E, D, G, L, A, Z, H, or apostrophe edit descriptor. The **field width** is the size of the field in characters.

The I, F, E, D, and G edit descriptors are collectively called **numeric edit descriptors** and are used to format integer, real, and complex data. The general rules that apply to these edit descriptors are:

On input

- Leading blanks are not significant. The interpretation of other blanks is controlled by the BLANK=**char_expr** specifier in the OPEN statement and the BN and BZ edit descriptors. A field of all blanks is considered to be zero. Plus signs are optional.

- With F, E, D, and G editing, a decimal point appearing in the input field overrides the portion of an edit descriptor that specifies the decimal point location. The field may have more digits than can be represented internally.

On output

- Characters are right-justified inside the field. Leading blanks are supplied if the editing process produces fewer characters than the field width. If the number of characters is greater than the field width, the entire field is filled with asterisks.

- A negative value is prefixed with a minus sign. By default, a positive or zero value is unsigned; however, it may be prefixed with a plus sign, as controlled by the S, SP, and SS edit descriptors.

¦ - On OS/400 and OS/2, a NaN (not a number) is indicated by question marks, plus infinity is indicated by plus signs, and minus infinity is indicated by minus signs.

Complex Editing: A complex value is a pair of separate real components. Therefore, complex editing is specified by a pair of F, E, D, or G edit descriptors. The first edit descriptor edits the real part of the number, and the second edit descriptor edits the imaginary part of the number. The two edit descriptors may be the same or different. One or more control (nonrepeatable) edit descriptors may appear between the two edit descriptors, but no data (repeatable) edit descriptors may appear between them.

Subtopics

12.3.1 / (Slash) Editing

Form:

```
/
```

The slash edit descriptor indicates the end of data transfer on the current record.

On input, when a file is connected for sequential access, the file is positioned at the beginning of the next record for each slash edit descriptor.

On output, when a file is connected for sequential access, a new record is created and the file is positioned to write at the start of the next record for each slash edit descriptor.

On input or output, when a file is connected for direct access, for each slash edit descriptor the record number increases by one, and the file is positioned at the beginning of the record that has that record number.

Examples of Slash Editing on Input

```
500  format(f6.2 / 2f6.2)
100  format(i4 / i4 / i4)
```

12.3.2 : (Colon) Editing

Form:

```
:
```

The colon edit descriptor terminates format control (which is discussed on page 12.2) if there are no more items in the input/output list. If there are more items in the input/output list when the colon is encountered, the colon is ignored.

Example of Colon Editing

```
10  format(3(:'Array Value',f10.5)/)
```

12.3.3 A (Character) Editing

Forms:

```
A
Aw
```

where:

w is an unsigned, nonzero, integer constant that specifies the width of the character field, including blanks. If **w** is not specified, the width of the character field is the length of the corresponding input/output list item.

The A edit descriptor directs the editing of character values. The A edit descriptor must correspond to an input/output list item of type character.

On input, if **w** is greater than or equal to the length (call it **len**) of the input/output list item, the rightmost **len** characters are taken from the input field. If the specified field width is less than **len**, the **w** characters are left-justified, with **len-w** trailing blanks added.

On output, if **w** is greater than **len**, the output field consists of **w-len** blanks followed by the **len** characters from the internal representation. If **w** is less than or equal to **len**, the output field consists of the leftmost **w** characters from the internal representation.

12.3.4 Apostrophe Editing

Form:

Same as a character constant (see page 3.11).

The apostrophe edit descriptor specifies a character string in an output format specification. The width of the output field is the length of the character constant.

Examples of Apostrophe Editing

```
50  format('The value is -- ',i2)
10  format(i2,'o''clock')
    write(*,'(i2,''o''''clock'')') itime
```

12.3.5 BN (Blank Null) and BZ (Blank Zero) Editing

Forms:

```
BN
BZ
```

The BN and BZ edit descriptors control the interpretation of nonleading blanks by subsequently-processed I, F, E, D, and G edit descriptors. BN and BZ have effect only on input.

BN specifies that blanks in numeric input fields are to be ignored, and remaining characters are to be interpreted as though right-justified. A field of all blanks has a value of zero.

BZ specifies that nonleading blanks in numeric input fields are to be interpreted as zeros.

The initial setting for blank interpretation is determined by the OPEN statement and its BLANK=**char_expr** specifier. (See page 11.5 for syntax.) The initial setting is determined as follows:

If OPEN is not specified, blank interpretation is system-dependent

If OPEN is specified but BLANK **char_expr** is not, blank interpretation is the same as if BN editing were specified.

If OPEN is specified and BLANK **char_expr** is specified, blank interpretation is the same as if BN editing were specified if the value of **char_expr** is NULL, or the same as if BZ editing were specified if the value of **char_expr** is ZERO.

The initial setting for blank interpretation takes effect at the start of execution of a formatted READ statement and stays in effect until a BN or BZ edit descriptor is encountered or until format control terminates. Whenever a BN or BZ edit descriptor is encountered, the new setting stays in effect until another BN or BZ edit descriptor is encountered, or until format control terminates.

12.3.6 E (Real with Exponent) and D (Double Precision) Editing

Forms:

Ew**.**d
Ew**.**d**E**e
Dw**.**d

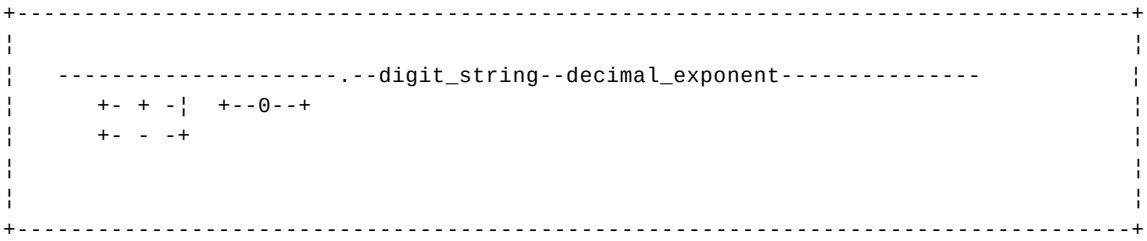
where:

w is an unsigned, nonzero, integer constant that specifies the width of the character field.
d is an unsigned integer constant that specifies the number of fraction digits to the right of the decimal point.
e is an unsigned, nonzero, integer constant that specifies the number of digits in the output exponent field. **e** has no effect on input.

The E and D edit descriptors direct editing between real and complex numbers in internal form and their character representations with exponents. An E or D edit descriptor must correspond to an input/output list item of type real, or to either part of an input/output list item of type complex.

The form of the input field is the same as for F editing.

The form of the output field for a scale factor of 0 is:



digit_string
is a digit string whose length is the **d** most significant digits of the value after rounding.

decimal_exponent
is a decimal exponent of one of the following forms (**z** is a digit):

Edit Descriptor	**Absolute Value of Exponent**	**Form of Exponent**
E**w.d**	|**decimal_exponent**| = 99	E±**z[1]z[2]**
E**w.d**	99 < |**decimal_exponent**| = 999	±z[1]z[2]z[3]
E**w.dEe**	|**decimal_exponent**| = (10**(e)**) - 1	E±**z[1]z[2]**...**z[e]**
D**w.d**	|**decimal_exponent**| = 99	D±**z[1]z[2]**

D**w.d**	99 < |**decimal_exponent**| = 999	±z[1]z[2]z[3]

The scale factor (**k**; see page 12.3.12) controls decimal normalization. If -**d** < **k**= 0, the output field contains |**k**| leading zeros and **d** - |**k**| significant digits after the decimal point. If 0 < **k** < **d** + 2, the output field contains **k** significant digits to the left of the decimal point and **d** - k + 1 significant digits to the right of the decimal point. Other values of **k** are not permitted.

See page 12.3 for general information about numeric editing.

Examples of E and D Editing on Input: (Assume BN editing is in effect for blank interpretation.)

Input	**Format**	**Value**
12.34	**e8.4**	**12.34**
.1234e2	**e8.4**	**12.34**
2.e10	**e12.6E1**	**2.e10**

Examples of E and D Editing on Output

Value	**Format**	**Output**
1234.56	**e10.3**	**0.123e+04**
1234.56	**d10.3**	**0.123d+04**

12.3.7 F (Real without Exponent) Editing

Form:

F**w.d**

where:

w is an unsigned, nonzero, integer constant that specifies the width of the character field.
d is an unsigned integer constant that specifies the number of fraction digits to the right of the decimal point.

The F edit descriptor directs editing between real and complex numbers in internal form and their character representations without exponents.

The F edit descriptor must correspond to an input/output list item of type real, or to either part of an input/output list item of type complex.

The input field for the F edit descriptor consists of, in order:

1. An optional sign.
2. A string of digits optionally containing a decimal point. If the decimal point is present, it overrides the **d** specified in the edit descriptor. If the decimal point is omitted, the rightmost **d** digits of the string are interpreted as following the decimal point and leading blanks are converted to zeros if necessary.
3. Optionally, an exponent, having one of the forms:
   A signed integer constant.
   E or D followed by zero or more blanks, followed by an optionally signed integer constant. E and D are processed identically.

The output field for the F edit descriptor consists of, in order:

1. Blanks if necessary
2. A minus sign if the internal value is negative, or an optional plus sign if the internal value is zero or positive
3. A string of digits that contains a decimal point and represents the magnitude of the internal value, as modified by the scale factor in effect and rounded to **d** fractional digits.

See page 12.3 for general information about numeric editing.

Examples of F Editing on Input: (Assume BN editing is in effect for blank interpretation.)

Input	**Format**	**Value**
-100	**f6.2**	**-1.0**
2.9	**f6.2**	**2.9**
4.e+2	**f6.2**	**400.0**

Examples of F Editing on Output

Value	**Format**	**Output**

+1.2	**f8.4**	**1.2000**
.12345	**f8.3**	**0.123**

12.3.8 G (General) Editing

Forms:

    G**w.d**
    G**w.dEe**

where:

w is an unsigned, nonzero, integer constant that specifies the width of the character field.
d is an unsigned integer constant that specifies the number of fraction digits to the right of the decimal point.
e is an unsigned, nonzero, integer constant that specifies the number of digits in the output exponent field.

The G edit descriptor is like the E and F edit descriptors except that the output format adapts to the magnitude of the number being edited. Thus the G edit descriptor provides a choice of output formats without requiring the magnitude of the numbers to be known ahead of time.

The G edit descriptor must correspond to an input/output list item of type real, or to either part of an input/output list item of type complex.

G input editing is the same as for F editing.

On output, the number is converted using either E or F editing, depending on the number. The field is padded with blanks on the right as necessary. Letting N be the magnitude of the number, editing is as follows:

    If N < 0.1 or N = 10 **d**):

    - G**w.d** editing is the same as E**w.d** editing
    - G**w.dEe** editing is the same as E**w.dEe** editing.

    If N = 0.1 and N < 10 **d**):

    - G**w.d** editing is the same as F**w'.d'** editing, where **w'** = **w** - 4 and **d'** = **d** - log[10]N
    - G**w.dEe** editing is the same as F**w'.d'** editing, where **w'** = **w** - (**e** + 2) and **d'** = **d'** - log[10]N.

See page 12.3 for general information about numeric editing.

Examples of G Editing on Output

Value	**Format**	**Output**
1234.56	**g12.5**	**1234.6**
123456.	**g12.5**	**0.12346e+06**

12.3.9 H Editing

Form:

nHh

where:

n is an unsigned, nonzero, integer constant that specifies the number of characters following the H, which make up the output field. Blanks are included in the count of characters.
h is a string of any of the characters permitted in a character constant (see page 3.11).

The H edit descriptor specifies a character string and its length in an output format specification.

If an H edit descriptor occurs within a character constant and includes an apostrophe, the apostrophe must be represented by two consecutive apostrophes, which are counted as one character in specifying **n**.

The H edit descriptor must not be used on input.

Examples of H Editing

```
    50  format(16hThe value is -- ,i2)
    10  format(i2,7ho'clock)
        write(*,'(i2,7ho''clock)') itime
```

12.3.10 I (Integer) Editing

Forms:

Iw
Iw.m

where:

w is an unsigned, nonzero, integer constant that specifies the width of the character field, including blanks and the optional sign.
m is an unsigned integer constant that specifies the minimum number of digits to be written. **m** must have a value that is less than or equal to **w**. **m** is useful on output only and has no effect on input.

The I edit descriptor directs editing between integers in internal form and character representations of integers. The corresponding input/output list item must be of type integer.

The input field for the I edit descriptor must be an optionally signed integer constant, unless it is all blanks (which is considered to be zero).

The output field for the I edit descriptor consists of, in order:

1. Zero or more leading blanks
2. A minus sign if the internal value is negative, or an optional plus sign if the internal value is zero or positive
3. The magnitude in the form of:

   If **m** is not specified, an unsigned integer constant without leading zeros.

   If **m** is specified, an unsigned integer constant of at least **m** digits and, if necessary, with leading zeros. If the internal value and **m** are both zero, blanks are written.

See page 12.3 for general information about numeric editing.

Examples of I Editing on Input: (Assume BN editing is in effect for blank interpretation.)

Input	Format	Value
-123	**i6**	**-123**
123456	**i7.5**	**123456**
1234	**i4**	**1234**

Examples of I Editing on Output

Value	Format	Output
-12	**i7.6**	**-000012**

12345	**i5**	**12345**

12.3.11 L (Logical) Editing

Form:

```
    Lw
```

where:

w is an unsigned, nonzero, integer constant that specifies the width of the character field, including blanks.

The L edit descriptor directs editing between logical values in internal form and their character representations. The L edit descriptor must correspond to an input/output list item of type logical.

The input field consists of optional blanks, followed by an optional decimal point, followed by a T for true or an F for false. Any characters following the T or F are accepted on input but are not acted upon; therefore, the strings .TRUE. and .FALSE. are acceptable input forms.

The output field consists of T or F preceded by **w**-1 blanks.

Examples of L Editing on Input

Input	**Format**	**Value**
t	**L4**	**true**
.false.	**L7**	**false**

Examples of L Editing on Output

Value	**Format**	**Output**
true	**L4**	**T**
false	**L1**	**F**

12.3.12 P (Scale Factor) Editing

Form:

kP

where:

k is the **scale factor**, an optionally-signed integer constant representing a power of ten.

The scale factor, **k**, applies to all subsequently-processed F, E, D, and G edit descriptors until another scale factor is encountered or until format control terminates. The value of **k** is zero at the beginning of execution of each input/output statement.

On input, when an input field using an F, E, D, or G edit descriptor contains an exponent, the scale factor is ignored. Otherwise, the internal value equals the external value multiplied by 10(-**k**).

On output:

With F editing, the external value equals the internal valu multiplied by 10(**k**).

With E and D editing, the external decimal field is multiplied b 10(**k**). The exponent is then reduced by **k**.

With G editing, fields are not affected by the scale factor unles they are outside the range that permits the use of F editing. If the use of E editing is required, the scale factor has the same effect as with E output editing.

Examples of P Editing on Input

Input	Format	Value
98.765	**3pf8.6**	**0.098765**
98.765	**-3pf8.6**	**98765.**
.98765e+2	**3pf10.5**	**.98765e+2**

Examples of P Editing on Output

Value	Format	Output
12.34	**2pf7.2**	**1234.00**
12.34	**-2pf6.4**	**0.1234**
12.34	**2pe10.3**	**12.34e+00**

12.3.13 S, SP, and SS (Sign Control) Editing

Forms:

```
S
SP
SS
```

The S, SP, and SS edit descriptors control the output of plus signs by all subsequently-processed I, F, E, D, and G edit descriptors until another S, SP, or SS edit descriptor is encountered or until format control terminates.

S and SS specify that plus signs are not to be written. (They produce identical results.) SP specifies that plus signs are to be written.

12.3.14 T, TL, TR, and X (Positional) Editing

Forms:

Tc
TL**c**
TR**c**
nX

where:

c is an unsigned, nonzero, integer constant.
n is an unsigned, nonzero, integer constant.

The T, TL, TR, and X edit descriptors specify the position at which the transfer of the next character to or from a record is to start. This position is:

For **c**, the **c**th character position

For T **c**, **c** characters backward from the current position. If the value of **c** is greater than or equal to the current position, then the next character accessed is position one of the record.

For T **c**, **c** characters forward from the current position.

For **n**X, **n** characters forward from the current position.

The TR and X edit descriptors give identical results.

On input, a TR or X edit descriptor may specify a position beyond the last character of the record if no characters are transferred from that position.

On output, a T, TL, TR, or X edit descriptor does not by itself cause characters to be transferred. If characters are transferred to positions at or after the position specified by the edit descriptor, positions skipped and not previously filled are filled with blanks. The result is the same as if the entire record were initially filled with blanks.

On output, a T, TL, TR, or X edit descriptor may result in repositioning such that subsequent editing with other edit descriptors causes character replacement.

Examples of T, TL, and X Editing on Input

```
  150  format(i4,t30,i4)
  200  format(f6.2,5x,5(i4,TL4))
```

Examples of T, TL, TR, and X Editing on Output

```
   50  format('Column 1',5x,'Column 14',tr2,'Column 25')
  100  format('aaaaa',TL2,'bbbbb',5X,'ccccc',T10,'ddddd')
```

12.3.15 Z (Hexadecimal) Editing

Form:

    Z**w**

where:

w is an unsigned, nonzero, integer constant that specifies the width of the character field, including blanks.

The Z edit descriptor directs editing between values of any type in internal form and their hexadecimal representation. (A hexadecimal digit is one of 0-9 or A-F.)

On input, **w** hexadecimal digits are edited and form the internal representation for the value of the input list item. The hexadecimal digits in the input field correspond to the rightmost hexadecimal digits of the internal representation of the value assigned to the input list item.

The output field contains **w** hexadecimal digits, including leading zeros. The digits in the output field correspond to the rightmost **w** hexadecimal digits of the internal representation.

Note that the editing of character data for input or output does not imply blank padding as it does for A editing.

Examples of Z Editing on Input

Input	**Format**	**Value**
0C	**z2**	**12**
7fff	**z4**	**32767**

Examples of Z Editing on Output

Value	**Format**	**Output**
12	**z4**	**000C**
-1	**z8**	**FFFFFFFF**

12.4 List-Directed Formatting

With **list-directed formatting**, editing is controlled by the types and lengths of the data being read or written. An asterisk format identifier specifies list-directed formatting. For example:

```
write(6,*) total1, total2
```

The characters in a formatted record processed under list-directed formatting constitute a sequence of values separated by value separators:

A value has the form of a constant or null value
A **value separator** is a comma, slash, or blank. A comma or slash may be preceded and followed by one or more blanks. Blanks in list-directed input records are significant.

Subtopics
12.4.1 List-Directed Input
12.4.2 List-Directed Output

12.4.1 List-Directed Input

Input list items in a list-directed READ statement are defined by corresponding values in records. The form of each input value must be acceptable for the type of the input list item. An input value is any of the following:

A value having the form of

- **constant**
- **r*constant**, where **r** is an unsigned, nonzero, integer constant. This form is equivalent to **r** successive appearances of the constant.

constant is an integer, real, double precision, complex, logical, or character constant.

A null value, represented by

- Two successive commas, with zero or more intervening blanks
- A comma followed by a slash, with zero or more intervening blanks
- An initial comma in the record, preceded by zero or more blanks.

More than one null value may be represented by the form **r***, where **r** is an unsigned integer constant. This form is equivalent to **r** successive null values.

A character value may be continued in as many records as required.

The end of a record:

Has the same effect as a blank unless the blank is within a characte value
Does not cause insertion of a blank or any other character in character value
Must not separate two apostrophes representing an apostrophe

Two or more consecutive blanks are treated as a single blank unless the blanks are within a character value.

A null value has no effect on the definition status of the corresponding input list item.

A slash marks the end of the input list, and list-directed formatting is terminated. If additional items remain in the input list when a slash is encountered, it is as if null values had been specified for those items.

12.4.2 List-Directed Output

List-directed WRITE and PRINT statements produce values in the order they appear in an output list. Values are written in a form that is reasonable for the data type of each output list item.

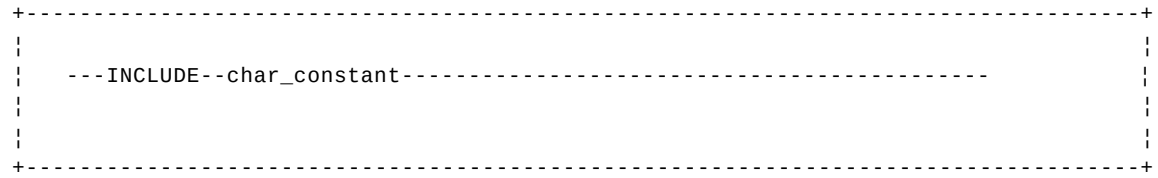
Logical values are written as T for the value true and F for the value false.

Character values are written as if the A edit descriptor were in effect. Character values written with list-directed output formatting cannot be read with list-directed input formatting because apostrophes are not written.

Slashes, as value separators, and null values are not written.

13.0 Chapter 13. INCLUDE Compiler Directive

MVS	VM	OS/400	OS/2
X	X	X	X



char_constant
is a character constant whose value, after trailing blanks have been removed, is a system-dependent file specifier naming the file to be included.

INCLUDE is a compiler directive. It directs the compiler to read source statements from an included file, which is a file different from the one containing the INCLUDE compiler directive.

When the compiler encounters an INCLUDE compiler directive, it suspends processing of the current file and continues with the first line of the included file. When the compiler reaches the end of the included file, it continues processing with the line following the INCLUDE compiler directive.

An INCLUDE compiler directive may appear anywhere in a FORTRAN source file.

An INCLUDE compiler directive must not be continued.

An included file may contain comment lines and complete FORTRAN statements, but must not contain other INCLUDE compiler directives.

A.0 Appendix A. Intrinsic Functions

Intrinsic functions are supplied by the FORTRAN processor. This appendix describes the intrinsic functions in Systems Application Architecture FORTRAN.

Some intrinsic functions may be referenced by a specific name, some by a generic name, and some by both. A **specific name** requires a specific argument type and produces a result of a specific type. A **generic name** does not require a specific argument type and usually produces a result of the same type as that of the argument. Generic names simplify the referencing of intrinsic functions.

Intrinsic Function	**Generic Name**	**Specific Name**	**Definition (See Notes)**	**No. of Arguments**	**Type of Argument**
Conversion to type integer	INT	--	y = sgn(x)	1	INTEGER*4
		INT	[|x|]		REAL*4 REAL*4
		IFIX			REAL*8
		IDINT			COMPLEX*8
		--			
			y = sgn		----------
	-------	-------	(re(z))		
			[|re(z)|]		REAL*4
		HFIX			
			----------	-----	
			y = sgn(x)		
			[|x|]		
Conversion to type real	REAL	REAL FLOAT -- SNGL -- DREAL	y = x	1	INTEGER*4
					INTEGER*4
					REAL*4 REAL*8
					COMPLEX*8
			y = re(z)		COMPLEX*16
Conversion to type double precision	DBLE	--		1	INTEGER*4
		--			REAL*4 REAL*8
		--			COMPLEX*8
		--	y = re(z)		
Conversion to type complex	CMPLX	--	y = x + i0,	1 or 2	INTEGER*4
		--	one argument		REAL*4 REAL*8
		--	y = x[1] +		COMPLEX*8
		--	ix[2], two		
			arguments		----------
	-------	-------			
			y = z		REAL*8
		DCMPLX			
			----------	-----	
			y = x + i0,		
			one argument		
			y = x[1] +		
			ix[2], two		
			arguments		
Truncation	AINT	AINT DINT	y = sgn(x)	1	REAL*4 REAL*8

			[|x|]		
Nearest whole number	ANINT	ANINT DNINT	y = sgn(x) [|x+.5|], x = 0 y = sgn(x) [|x-.5|], x < 0	1	REAL*4 REAL*8
Nearest integer	NINT	NINT IDNINT	y = sgn(x) [|x+.5|], x = 0 y = sgn(x) [|x-.5|], x < 0	1	REAL*4 REAL*8
Absolute value	ABS	IABS ABS DABS CABS CDABS	y = |x| 'y = |z| = (re(z)' sup 2 ' + im(z)' sup 2 ')' sup <1/2>	1	INTEGER*4 REAL*4 REAL*8 COMPLEX*8 COMPLEX*16
Remaindering	MOD	MOD AMOD DMOD	y = x[1] - [x[1]/x[2]] x[2]	2	INTEGER*4 REAL*4 REAL*8
Transfer of sign	SIGN	ISIGN SIGN DSIGN	y = sgn(x[2]) |x[1]|	2	INTEGER*4 REAL*4 REAL*8
Positive difference	DIM	IDIM DIM DDIM	y = x[1] - x[2], x[1] > x[2] y = 0, x[1] = x[2]	2	INTEGER*4 REAL*4 REAL*8
Double precision product		DPROD	y = x[1] x[2]	2	REAL*4
Choosing largest value	MAX ------- 	MAX0 AMAX1 DMAX1 ------- AMAX0 MAX1	y = max(x[1], . . . x[n])	= 2	INTEGER*4 REAL*4 REAL*8 ------------ INTEGER*4 REAL*4
Choosing smallest value	MIN ------- 	MIN0 AMIN1 DMIN1 ------- AMIN0 MIN1	y = min(x[1], . . . x[n])	= 2	INTEGER*4 REAL*4 REAL*8 ------------ INTEGER*4 REAL*4
Imaginary part of a complex	IMAG	AIMAG DIMAG	y = im(z)	1	COMPLEX*8 COMPLEX*16

argument					
Complex conjugate	CONJG	CONJG DCONJG	y = z(*) if z = a + ib, then z(*) = a - ib	1	COMPLEX*8 COMPLEX*16
Square root	SQRT	SQRT DSQRT CSQRT CDSQRT	'y = x' sup <1/2> 'y = z' sup <1/2>	1	REAL*4 REAL*8 COMPLEX*8 COMPLEX*16
Exponential	EXP	EXP DEXP CEXP CDEXP	'y = e' sup x 'y = e' sup z	1	REAL*4 REAL*8 COMPLEX*8 COMPLEX*16
Natural logarithm	LOG	ALOG DLOG CLOG CDLOG	y = log[e](x), x > 0 y = log[e](z), z &ne. 0 + i0	1	REAL*4 REAL*8 COMPLEX*8 COMPLEX*16
Common logarithm	LOG10	ALOG10 DLOG10	y = log[10]x, x > 0	1	REAL*4 REAL*8
Sine	SIN	SIN DSIN CSIN CDSIN	y = sin(x) y = sin(z)	1	REAL*4 REAL*8 COMPLEX*8 COMPLEX*16
Cosine	COS	COS DCOS CCOS CDCOS	y = cos(x) y = cos(z)	1	REAL*4 REAL*8 COMPLEX*8 COMPLEX*16
Tangent	TAN	TAN DTAN	y = tan(x)	1	REAL*4 REAL*8
Arcsine	ASIN	ASIN DASIN	y = arcsin(x) |x| = 1, -&pi./2 = y = &pi./2	1	REAL*4 REAL*8
Arccosine	ACOS	ACOS DACOS	y = arccos(x) |x| = 1, 0 = y = &pi.	1	REAL*4 REAL*8
Arctangent	ATAN ------- ATAN2	ATAN DATAN ------- ATAN2 DATAN2	y = arctan(x) - &pi./2 = y = &pi./2 ---------- Computes angle y such that ' sin y = x' sub 2 '/(x' sub 1 '' sup 2	1 ----- 2	REAL*4 REAL*8 ---------- REAL*4 REAL*8

			'+ x' sub 2 '' sup 2 ')' sup <1/2> ',' ' cos y = x' sub 1 '/(x' sub 1 '' sup 2 '+ x' sub 2 '' sup 2 ')' sup <1/2> x[1] &ne. 0 and x[2] &ne. 0 -&pi. < y = &pi.		
Hyperbolic sine	SINH	SINH DSINH	'y = ' <'e' sup 'x' - 'e' sup <- 'x'>> over 2 |x| < 175.366	1	REAL*4 REAL*8	
Hyperbolic cosine	COSH	COSH DCOSH	'y = ' <'e' sup 'x' + 'e' sup <- 'x'>> over 2 |x| < 175.366	1	REAL*4 REAL*8	
Hyperbolic tangent	TANH	TANH DTANH	'y = ' <'e' sup 'x' - 'e' sup <- 'x'>> over <'e' sup 'x' + 'e' sup <- 'x'>>	1	REAL*4 REAL*8	
Conversion to type integer		ICHAR	Position of x in the collating sequence	1	CHARACTER*1	
Conversion to type character		CHAR	Character corresponding to position of x in the collating sequence	1	INTEGER*4	
Length		LEN	Length of x	1	CHARACTER	
Index of a substring		INDEX	Location of substring x[2] in string x[1]	2	CHARACTER	
Lexically greater than or equal		LGE	x[1] = x[2] Comparison is ASCII	2	CHARACTER	

Lexically greater than		LGT	x[1] > x[2] Comparison is ASCII	2	CHARACTER
Lexically less than or equal		LLE	x[1] = x[2] Comparison is ASCII	2	CHARACTER
Lexically less than		LLT	x[1] < x[2] Comparison is ASCII	2	CHARACTER
Inclusive or		IOR	y = or(x[1],x[2])	2	INTEGER*4
Logical product		IAND	y = and(x[1],x[2])	2	INTEGER*4
Logical complement		NOT	y = not(x[1])	1	INTEGER*4
Exclusive or		IEOR	y = xor(x[1],x[2])	2	INTEGER*4
Shift operations		ISHFT	x[1] is shifted by x[2] bits to right if x[2] < 0 or to left if x[2] > 0, where |x[2]| = 32	2	INTEGER*4
Bit test		BTEST	y = true if bit x[2] of x[1] = 1 or false if bit x[2] of x[1] = 0	2	INTEGER*4
Bit set		IBSET	y = bitset(x[1],x[2]) sets bit x[2] of x[1] to 1	2	INTEGER*4
Bit clear		IBCLR	y = bitclear(x[1],x[2]) sets bit x[2] of x[1] to 0	2	INTEGER*4

Notes about Definitions:

Definitions use familiar mathematical function names, which have their mathematical meanings or are defined below.

The bits in bit-manipulation function BTEST, IBSET, and IBCLR are numbered

from right to left, beginning at zero.

The result of a function of type complex is the principal value.

Meanings of symbols:

x   denotes a single argument.
x[i] denotes the i-th argument when a function accepts more than one
     argument.
[x] denotes the integer part of the number x.
sgn(x) is +1 if x = 0 or -1 if x < 0.
y   denotes a function result.
z   denotes a complex argument.

B.0 Appendix B. Compiler Considerations

The following compiler options affect your program's conformance to Systems Application Architecture:

For MVS and VM (VS FORTRAN Version 2)

Use the OCSTATUS execution option.

For OS/400 (FORTRAN/400)

Do not use the *F66, *SHORTI, and *NOSAVE compiler options.

For OS/2 (FORTRAN/2)

Do not use the /F and /I compiler options.

CHANGES Summary of Changes

This edition adds IBM FORTRAN/400 (Program Number 5730-FT1) as the product which implements SAA CPI FORTRAN in the OS/400 environment. Text that describes the new FORTRAN/400 support is indicated by a vertical bar to the left of the changes.

In addition, minor technical and editorial changes have been made throughout.

A

B

F

T